

Advanced AAA

As Chapter 16, “AAA,” shows, the DataPower AAA framework is flexible and powerful, allowing you unparalleled control over which users or systems can access your services. This chapter goes through some examples of this at work, including customization of the framework and integration with IBM Tivoli Access Manager (TAM).

Customization is an inherent part of the DataPower device and isn’t something to be afraid of. While there are many out-of-the-box options that cover the vast majority of customer requirements, especially in a complex area such as security, there is no way that everything can be covered. The AAA framework enables you to customize to a granular level to integrate with even the most complex of solutions and fulfill even the most difficult requirements.

Customizing the AAA Runtime Process

The AAA runtime described in Chapter 16 can be customized at almost every step. (The exception is resource extraction, but this can be fixed by a subsequent customized resource mapping step.) The steps of the AAA flow are summarized in Figure 17-1. To write a AAA customization stylesheet, you need to know what the inputs and outputs of each stage need to look like.

XML in the AAA Flow

The AAA flow works by passing an XML tree from one step to the next; the XML tree contains information about what happened in the earlier stages. It is built automatically by the out-of-the-box AAA methods; some of these XML trees were shown in Chapter 16 as the contents of the debug variables for the various stages of AAA processing.

When customizing the AAA flow, our own stylesheets need to work with this XML tree and create the relevant output for the next stage. This may seem somewhat complex, but in truth is no different to writing any other XSL; our processing must simply transform the input tree from one stage to the output tree of another, and by doing so, it communicates decisions to DataPower regarding authentication, authorization, and so on.

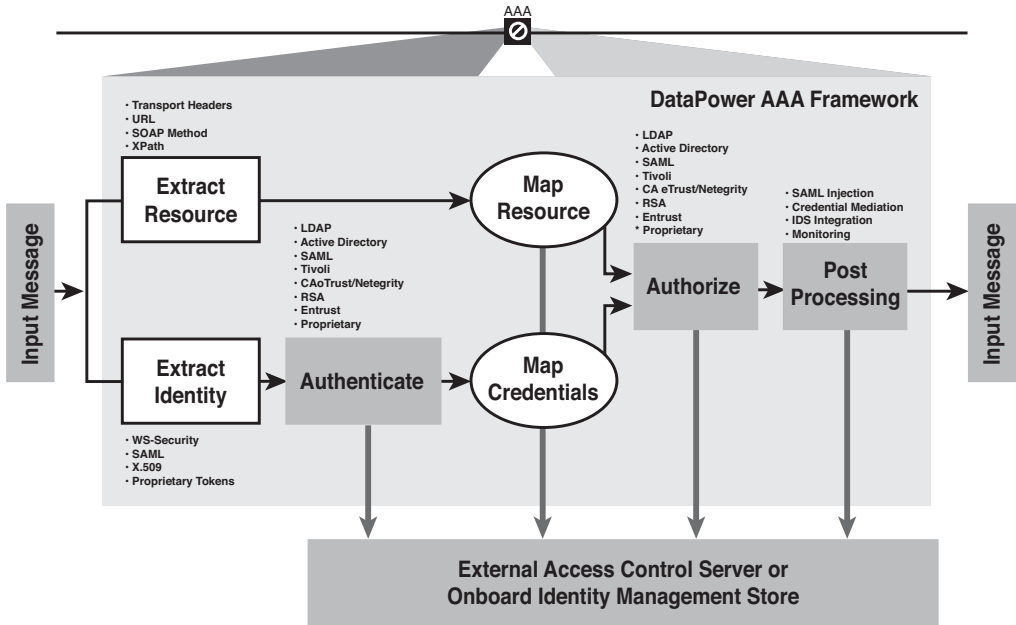


Figure 17-1 AAA flow summary.

This processing is best demonstrated by example, so we will walk you through each stage and show an example of processing at each level. It is highly likely that when you do need to customize the AAA process, you will have to customize only one or two stages; our approach, therefore, enables you to pick the pieces you need to use to customize in your specific environment.

EI—Extract Identity

The EI stage has many out-of-the-box options that cover the vast majority of user requirements. Indeed, it is difficult to think of a realistic requirement that is not already covered! Of course there are real-world requirements, but they do not make for an easy demonstration of the functionality, so this example may appear somewhat contrived; we can only apologize for the huge array of supported out-of-the-box use cases!

The example we use is that of an XML document that contains a username and a password in a specific node. For most similar use cases, we would use the out of the box processing called Token Extracted from the Message, which allows you to specify an XPath expression to extract the nodeset containing the part of the incoming XML document that you wish to treat as a token. However, in this instance the username field needs to be modified before passing to the authenticate step, and therefore, we need to use custom processing.

The input for the EI stage custom processing is the entire XML message passed from the input context to the AAA action. The entire request is available, including protocol headers, through DataPower extension functions. It may be that other processing has taken place before the AAA action; for instance, the client XML message may be encrypted and a Decrypt action

was needed, or perhaps form-based login was used in a Web application and a transformation was required to convert it from HTTP to XML. For our example, however, no previous processing has taken place, and the request document appears as shown in Listing 17-1.

Listing 17-1 Request Document

```
<?xml version="1.0"?>
<getAmount>
  <authentication>
    <username>alice</username>
    <password>mysecret</password>
  </authentication>
  <data>
    <values/>
  </data>
</getAmount>
```

Clearly the username and password need to be extracted from the request; however, in this instance, this is not enough, because our authentication server requires usernames to be in the form of an email address to uniquely identify specific users. (There is more than one alice in the world, for example.) The domain part of the username is passed in using an HTTP header, AuthDomain, the value of which needs to be appended to the supplied username in order to form an email address.

This is accomplished using the stylesheet shown in Listing 17-2. If you are not familiar with XSLT, a wealth of information about developing custom stylesheets can be found in Part VI of this book, “DataPower Development.”

Listing 17-2 Identity Extraction Stylesheet—IE.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dp="http://www.datapower.com/extensions"
  exclude-result-prefixes="dp">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <!-- Use /*/ to match various requests -->
    <xsl:variable name="username"
      select="/*/authentication/username/text()"/>
    <xsl:variable name="password"
      select="/*/authentication/password/text()"/>
    <xsl:variable name="domain"
      select="dp:request-header('AuthDomain')"/>
    <username><xsl:value-of
      select="concat($username,'@',$domain)"/></username>
    <password sanitize="true"><xsl:value-of
      select="$password"/></password>
  </xsl:template>
</xsl:stylesheet>
```

This stylesheet processes the input document directly, searching for the username and password fields. It also retrieves the value of the AuthDomain HTTP header. Finally, it outputs an XML structure containing username and password elements with the relevant values for authentication. This choice of output is not accidental; the expected output in this instance is two XML nodes—username and password—with no wrapper element.

The stylesheet is configured in the EI step of a AAA policy, as shown in Figure 17-2.

DATAPOWER XI50

Configure an Access Control Policy

AAA Policy Name: custom_aaa

Define how to extract a user's identity from an incoming request.

Identification Methods

- ☐ HTTP Authentication Header
- ☐ Password-carrying UsernameToken Element from WS-Security Header
- ☐ Derived-key UsernameToken Element from WS-Security Header
- ☐ BinarySecurityToken Element from WS-Security Header
- ☐ WS-SecureConversation Identifier
- ☐ WS-Trust Base or Supporting Token
- ☐ Kerberos AP-REQ from WS-Security Header
- ☐ Kerberos AP-REQ from SPNEGO Token
- ☐ Subject DN of the SSL Certificate from the Connection Peer
- ☐ Name from SAML Attribute Assertion
- ☐ Name from SAML Authentication Assertion
- ☐ SAML Artifact
- ☐ Client IP Address
- ☐ Subject DN from Certificate in the Message's signature
- ☐ Token Extracted from the Message
- ☐ Token Extracted as Cookie Value
- ☐ LTPA Token
- ☐ Processing Metadata
- ☒ Custom Template

URL: local:/// EI.xsl Upload... Fetch...

Back Next Cancel

Figure 17-2 Select a custom template and provide the XSL file.

We can use the Probe as described in Chapter 16 to examine the execution trace of the AAA action and see the XML nodeset on the input and output of each custom transformation. If we enable a Probe, submit a transaction, refresh the Probe, and click on that transaction, we are shown a step-by-step representation of the various stages of processing that the request went through. We can click on each stage, in turn, to get details; if we click on the AAA action in the policy, the first tab titled Extension Trace gives details of the custom processing in the AAA policy. As shown in Figure 17-3, each stage has a request and a response nodeset. We can click on the link next to each one to get details of the input or output nodeset for the next part of the processing.

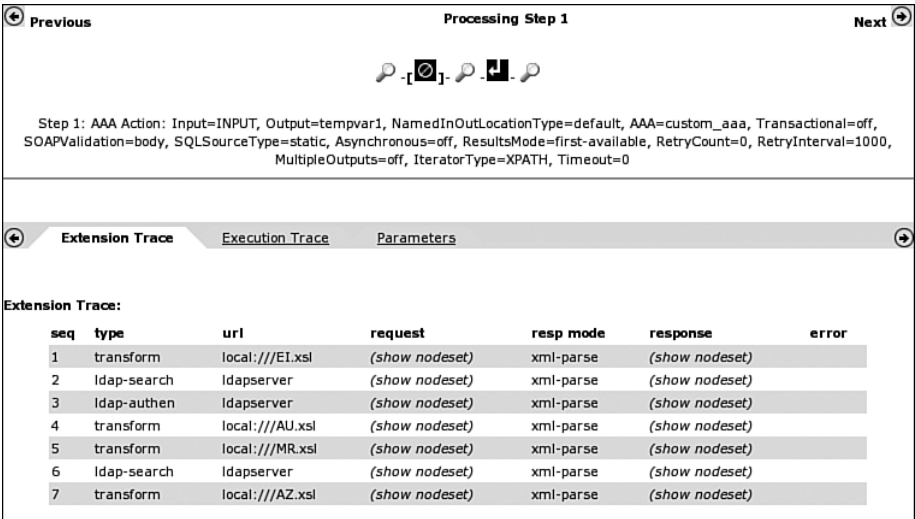


Figure 17-3 The processing as seen in the Probe.

For instance, Figure 17-4 shows the input into the EI step in the Probe, retrieved by clicking the (show nodeset) link in the request column, which confirms that the input to the EI step is the full submitted XML message.

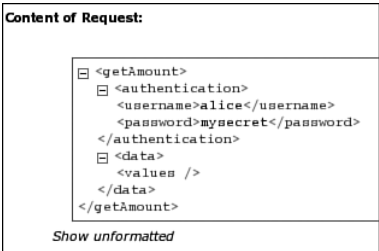


Figure 17-4 Input into the EI step.

This is then transformed by our stylesheet, which also includes the value of the HTTP header AuthDomain, and the output is shown in Figure 17-5.

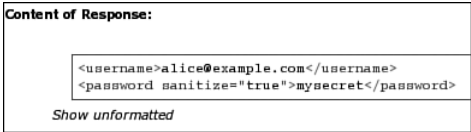


Figure 17-5 Output from the EI step.

The output from our custom processing presents simply the extracted identity, which Data-Power wraps in an XML structure and passes to the next step.

TIP—SANITIZE PASSWORDS

The previous example shows a construct containing the password, and an attribute has been added to the password node, `sanitize="true"`. By adding this special attribute, we are telling the appliance that the text contents of this node should be masked in any log messages that the device happens to display. For instance, even with debug level AAA logging enabled, the device replaces the actual password supplied with asterisks. This ensures that, when the logs are stored “off box” in a persistent manner (as they always should be in a production environment), the password information is not stored in an insecure location that is not directly under the access control of the device.

Note that this sanitization does not apply to the Probe, as should be clear from Figure 17-2. This should not be cause for concern because the Probe can only be enabled on the appliance itself, and even then only by a user with authority to modify the object being probed. In addition, the Probe should not be used in production (as the warning on the administrative panel when enabling the Probe states clearly).

The XML output from our EI stylesheet is used as the input to the AU step, so the exact content of the response is passed as the input. DataPower automatically wraps this in an XML structure identifying it as an `entry` inside an `identity`, and the entry has an attribute called `type` with the value `custom` and a second attribute called `url` showing where the custom stylesheet is stored. This information is shown in Figure 17-6, which shows the input into the AU step, again taken from the extension trace in the Probe.

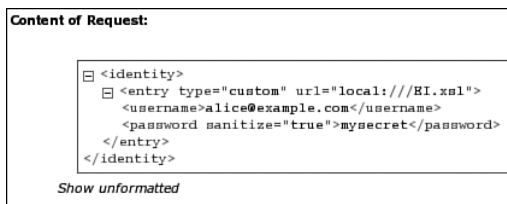


Figure 17-6 Output from EI is also input to AU.

The identity XML tree can have more than one entry node, depending on how many options were selected when defining the policy. Thus, if we were to select IP address as well as custom, we would have two entry nodes added by DataPower: one of type `custom` and the second of type `client-ip-address`, as shown in Figure 17-7.

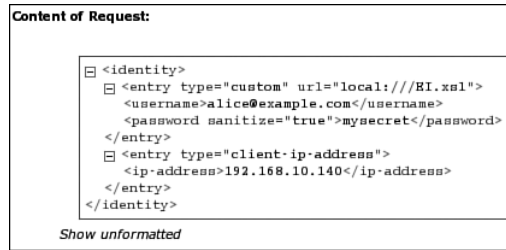


Figure 17-7 It is possible to have more than one entry as the output from EI.

However, be aware that the contents can conflict with each other and lead to unexpected results. For instance, because we have created a username and password node, it is likely that our next step, AU, will be configured to use that username and password node to authenticate the user. If we were to *also* select HTTP basic authentication for EI, things would work just fine as long as *either* the custom stylesheet *or* the basic authentication header contained a valid username and password combination. However, if the user then provided *both* a HTTP basic authentication header *and* a username/password for our custom XSL to extract identity from, there would be two sets of username and password nodes. In that situation, because there is no way to tell which set to use, the device will use neither and simply deny the request, as shown in Figure 17-8 (taken from the DataPower log).

```

xmloffice (custom_aaa): Rejected by filter; SOAP fault sent
xmloffice (custom_aaa): request custom_aaa_request #1 aaa: 'INPUT custom_aaa stored in tempvar1' failed: Username mismatch in input
xmloffice (custom_aaa): Execution of 'store:///dp/aaapolicy.xml' aborted: Username mismatch in input
xmloffice (custom_aaa): Policy(custom_aaa): Message rejected
xmloffice (custom_aaa): Reject set: Username mismatch in input
xmloffice (custom_aaa): Policy(custom_aaa): Username mismatch in input
xmloffice (default): xslt Compilation Request: Found in cache (local:///EI.xsl)
xmloffice (default): xslt Compilation Request: Checking cache for URL local:///EI.xsl
xmloffice (custom_aaa): Policy(custom_aaa): Extracting identity using "http-basic-auth+custom"
xmloffice (default): xslt Compilation Request: Found in cache (store:///dp/aaapolicy.xml)

```

Figure 17-8 Supplying identical identity nodes with two types will fail.

Incidentally, if the usernames are the same, at least processing continues to the next stage, although this may not be what you want, especially if a different password is specified for each!

AU—Authenticate

The AU stage takes the input of the EI stage and processes the extracted identity such as to authenticate it; that is, to ensure in some manner that a valid set of credentials have been presented to prove that the peer entity supplying the credentials actually is the identity they claim to be.

WARNING—SECURITY PROGRAMMING IS HARD

It is easy to accidentally write an AU stylesheet that works and yet is completely insecure. Customization of the AAA process needs to be done with care and attention because this is security code that defines fundamental access control for your system, but, in particular, the AU stylesheet is easy to write badly and should be reviewed and validated many times.

At the AU stage, your stylesheet must decide a basis on which to either trust or not trust the supplied credentials. If this trust is too weak, people will be able to abuse the trust to access resources they should not be able to access. For example, if the stylesheet simply looks at an HTTP header to determine whether the request comes from a specific trusted server, a malicious client could simply spoof this header and become authenticated. If on the other hand the header contains an encrypted signed short lifetime token containing a username, malicious clients would find it much harder to compromise.

An important point for authentication in particular is that successful authentication is signified from an AU stylesheet by outputting any valid XML. Failed authentication is shown by outputting *no* XML at all. It is key to understand this; if your stylesheet outputs any valid XML, authentication is considered to have succeeded, even if the node set looks like this:

```
<authentication>
  <status>FAILED</status>
</authentication>
```

This XML would actually be seen as successful authentication if output by a custom AU stylesheet! To signify authentication failure, you must output nothing.

So, let's take a look at an example AU stylesheet. This stylesheet takes as its input the username and password from the previous example. Recall that the username was made up in the form of an email address, and the domain of the email address was referred to as an authentication domain. Our goal in this AU stylesheet is to authenticate this user to an LDAP server, using a different base distinguished name depending on the domain provided.

The LDAP server works by looking up the email address provided by the client and resolving that into a specific distinguished name; that distinguished name is then used to bind to validate that the password is correct. The AU.xml stylesheet that performs this interaction with LDAP is shown in Listing 17-3.

Listing 17-3 AU Stylesheet Communicating with LDAP—AU.xml

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dp="http://www.datapower.com/extensions"
  exclude-result-prefixes="dp">
<xsl:output method="xml"/>
```

```

<xsl:template match="/">
<!-- Take the username password and domain from the incoming nodeset -->
<xsl:variable name="username"
  select="/identity/entry[@type='custom']/username/text()" />
<xsl:variable name="password"
  select="/identity/entry[@type='custom']/password/text()" />
<xsl:variable name="domain"
  select="substring-after($username, '@')"/>
<xsl:choose>
  <xsl:when test="$domain=('example.com' or 'somewhere.com')">
    <xsl:variable name="SearchFilter"
      select="concat('& (objectClass=ePerson) (emailAddress=',
        $username,')')"/>
    <!-- make a SUB search for our email address to find the DN -->
    <xsl:variable name="userid"
      select="dp:ldap-search('ldapservers', '389', '', '',
        concat('o=', $domain), 'dn', $SearchFilter, 'SUB', '', '')"/>
    <!-- if the search returned a distinguished name... -->
    <xsl:if test="$userid/LDAP-search-results/result/DN/text()">
      <!-- ..then try and bind as that distinguished name -->
      <xsl:if test="dp:ldap-authen(
        $userid/LDAP-search-results/result/DN/text(),
        $password, 'ldapservers:389')">
        <!-- output XML if successful -->
        <authenticated>yes</authenticated>
      </xsl:if>
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <!-- OUTPUT NOTHING - AU fails -->
  </xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

Let's examine this processing in slightly more depth. This is an LDAP server with two base distinguished names: example.com and somewhere.com. To perform the initial search, we first build an RFC2254-compliant LDAP filter string that looks for an LDAP object with objectClass of ePerson (a common object class representing a user) and an email address equal to that supplied (which was built, recall, by adding the username to the AuthDomain HTTP header). This search filter is then used in an ldap-search extension function call, to contact the server at host-name ldapservers on port 389, and perform an LDAP search for a subset of the relevant base Distinguished Name and extract the DN field specific to that user. Figure 17-9 shows the input into the ldap-search call, again taken from the extension trace in the Probe.

Content of Request:

```

<parameter>
  <bindDN />
  <bindPassword />
  <lookupDN>o=example.com</lookupDN>
  <lookupAttribute>dn</lookupAttribute>
  <filter>(&(objectClass=Person)(emailAddress=alice@example.com))</filter>
</parameter>

```

Show unformatted

Figure 17-9 The input nodeset to the ldap-search call.

This nodeset is built automatically by DataPower as a result of running the ldap-search extension function. As is clear from Figure 17-9, the actual connection to LDAP for the search is anonymous. If the server did not support anonymous binds for searching, we could specify a DN and password to use in the ldap-search function call. The search does not authenticate the user at all; rather it finds the attribute in the lookupAttribute node, in this case the DN, shown in Figure 17-10.

Content of Response:

```

<LDAP-search-results>
  <result>
    <DN>uid=alice.smith,c=gb,ou=sales,o=example.com</DN>
  </result>
</LDAP-search-results>

```

Show unformatted

Figure 17-10 The nodeset containing the result of the LDAP search.

After we have the distinguished name, we then attempt to bind as that user with the password provided, by using the ldap-authn function call. The ldap-authn input parameters in this instance are shown in Figure 17-11.

Content of Request:

```

<parameter>
  <bindDN>uid=alice.smith,c=gb,ou=sales,o=example.com</bindDN>
  <bindPassword>mysecret</bindPassword>
  <lookupDN />
  <lookupAttribute />
  <filter />
</parameter>

```

Show unformatted

Figure 17-11 The nodeset built by ldap-authn.

Again, this nodeset is automatically built by the ldap-authn function call. If this bind fails, the ldap-authn call returns no XML nodeset. If the bind is successful, however, the response of the ldap-authn call will contain the DN that was authenticated, depicted in Figure 17-12.

Content of Response:

```
<entry type="ldap">uid=alice.smith,c=gb,ou=sales,o=example.com</entry>
```

Show unformatted

Figure 17-12 The response nodeset from the ldap-authen call.

Our stylesheet simply needs to output any valid XML in order for authentication to have been deemed “successful.” However, it is often useful to be able to pass on a credential of some sort to the rest of the AAA policy, such that it can make authorization decisions based on that credential. The output of the AU phase, whatever we include, will be made available to later phases in AAA processing, so we can choose to, for instance, pass in the DN that was authenticated. The structure we create can be seen in the Probe either as the output nodeset of our XSLT or by examining the AU debug context variable as shown in Figure 17-13.

Content of Variable 'var://context/AAA-internal-debug/AU':

```
<credentials>
  <entry type="custom" url="local:///AU.xml">
    <authenticated>
      <entry type="ldap">uid=alice.smith,c=gb,ou=sales,o=example.com</entry>
    </authenticated>
  </entry>
</credentials>
```

Show unformatted

Figure 17-13 The final output of our AU stylesheet.

This XML structure has been generated by DataPower to wrap the output of our stylesheet, and it will be passed to the Map Credentials step.

Because Web services are inherently stateless, there likely will be multiple inbound requests with the same credentials. For performance reasons, the results of the authentication step, as well as the authorization step, will by default be cached for three seconds. Caching can be just as useful for a custom AAA step, because it will lower the load of any backend systems that the AAA step connects with. However, care should be taken that this caching does not cause undesired effects. For instance, if your custom AU stylesheet connects to a service that maintains an audit log intended to record all authentications, caching would mean that some cached requests would not be recorded, and thus caching should likely be disabled. The cache lifetime is configurable within the definition of the AAA policy, and it can be completely disabled if required.

WARNING—Do Not Rely on the Probe Debug Variables

Figure 17-13 shows the content of the internal Probe variable `var://context/AAA-internal-debug/AU`, instead of using the extension trace as previous examples have. This is deliberate in order to raise this important point: Although these variables are extremely useful for debugging, they should *not* be relied upon in your XSLT!

These variables exist only when the Probe is enabled. This means that, if in your XSLT you read the contents of this or any other of the AAA-internal-debug variables, and make processing decisions based on them, your code will not work when the Probe is disabled.

The Probe should always be disabled in production environments, which means that testing must be carried out much earlier with the Probe disabled, just to make sure that nothing you are doing is affected by the Probe.

MC—Map Credentials

The Map Credentials (MC) step, as discussed in Chapter 16, enables you to modify the output of the authenticate step using a number of methods; one of those methods is to run a custom stylesheet. Of course, when the previous step of authentication was carried out using a stylesheet, in which we were able to present the output of the AU step in any way we chose, there is little value in further modifying it using a second stylesheet in the MC step! Thus, in most instances when AU uses a stylesheet, MC will not.

If authentication was performed using an out-of-the-box method, however, it can often be beneficial to use a custom stylesheet to perform credential mapping. This is also a nice way of performing further authentication over and above what the out-of-the-box authentication methods provide. For instance, if AU was used to simply bind a DN and password to verify that it is valid (a common use case), the MC step could take these credentials and perform further validation (for instance ensure that the user is not on a specific list of users who should be barred from logging in) or enrichment (such as pulling data out of a database for the specific user which will be used during a later phase such as authorization) or token exchange (for instance via a call out to Tivoli Federated Identity Manager).

One specific example of a credential mapping step that is often required is to map a distinguished name from x.509 format, as may often be found in a digital certificate, to the format more commonly found in LDAP servers. For example, the distinguished name `uid=user1, ou=research, o=example.com` would likely be presented in a certificate in x.509 format as `o=example.com/ou=research/uid=user1`—that is, slashes are used instead of commas and the order is reversed. To use this DN to check against an LDAP server, we would need to map the entries into the reverse order; a stylesheet to perform this mapping is shown in Listing 17-4.

Listing 17-4 MC Stylesheet to Map Distinguished Names—MC.xsl

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dp="http://www.datapower.com/extensions"
  xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
  xmlns:str="http://exslt.org/strings"
  extension-element-prefixes="dp" exclude-result-prefixes="dp"
  version="1.0">
  <xsl:output method="xml" />
  <xsl:template match="/">
    <xsl:variable name="originalDN"
      select="/credentials/entry[@type='validate-signer']
        /CertificateDetails/Subject/text()" />
    <xsl:variable name="splitDN">
      <xsl:copy-of select="str:split($originalDN, '/')" />
    </xsl:variable>
    <xsl:variable name="newSplitDN">
      <xsl:for-each select="$splitDN/token">
        <xsl:sort order="descending" data-type="number"
          select="position()" />
        <xsl:variable name="this"
          select="normalize-space()" />
        <xsl:value-of select="concat(' ', $this)" />
      </xsl:for-each>
    </xsl:variable>
    <xsl:variable name="newDN"
      select="substring-after($newSplitDN, ' ')" />
    <entry type='custom'>
      <dn><xsl:value-of select="$newDN" /></dn>
    </entry>
  </xsl:template>
</xsl:stylesheet>

```

If the MC stage returns no credentials, the authorization step is likely to fail. Of course, whether or not it does is customizable! The default authorization processes consider authentication to have passed only if credentials are returned by the MC stage; in particular, this is true for the “any authenticated” authorization option, which is perhaps not obvious.

Read that again. The default authorization processes consider authentication to have passed only if credentials are passed to them. What this means is that, if your MC stage changes your authentication, strange and unexpected things can happen. An especially bad case might be if authentication fails, but your credential mapping still outputs a credential; in this case, a user could pass authorization and be allowed to access the protected resource they are trying to, even though authentication failed! Of course, this would likely be due to a bug in your stylesheet, but that is scant comfort when the villains are making off with the swag.

There may however be good use cases for changing the credentials to be valid after a failed authentication. For example, it is possible that if someone fails authentication, we might still want to allow them access to the system but as a guest user with limited privileges.

ER—Extract Resources

The ER stage is unique among the AAA stages in that it is the only one that cannot be replaced with a custom stylesheet. The output of the ER stage can, however, be modified or even completely removed by using a custom stylesheet in the MR stage which is next.

Resource extraction can be flexible. It can include processing metadata, the power of which was shown in Chapter 16, an XPath expression to extract any specific piece of the incoming XML message, and a number of other options as shown in Figure 17-14.

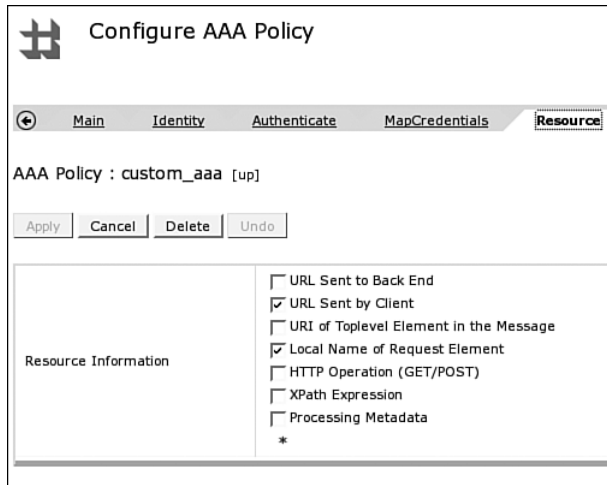


Figure 17-14 Resource extraction can be flexible.

A node tree is output by the ER stage that is used as the input to the MR stage that follows. For the example shown in Figure 17-14, with two types of resources extracted, this node tree looks like Figure 17-15.

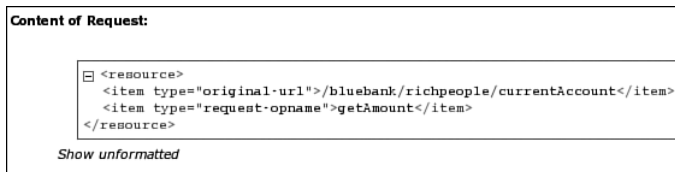


Figure 17-15 Multiple resources.

MR—Map Resources

The Map Resources stage modifies the extracted resource. Why would we want to do that? The most common use is to add together different resources of different types into a single resource of a specific type that an external authorization system will be able to understand.

Let's say that our external authorization system, as many do, works by protecting URLs. It can express an authorization constraint in terms of URL pattern matching; for instance, it would be able to protect the following:

```
/bluebank/richpeople/currentaccount
```

This happens to be the URI of a Web service served on a backend application server. Our external authorization system can state a constraint that, in order to access this URL, you have to have authenticated. However, as far as the external authorization system is concerned, this is only a URL; it has no concept of Web services.

This is unfortunate, because under the current account Web service URL shown, our backend application server provides a number of different Web service operations. It has an addMoney operation, a removeMoney operation, and a getAmount operation. What if we want to give one group of users the ability to run just the getAmount operation, and another group of users the ability to addMoney and removeMoney? Or, for instance, what if we require one level of authentication to getAmount, but require extra authentication to addMoney and removeMoney? These are real requirements; authorization at the operation level is a common thing to want to do.

DataPower, of course, is flexible and can easily protect at the Web service operation level. But if the external authorization system cannot understand Web service operations, how can we even ask it who should be allowed to call which operation? That's where the Map Resources step shines. The stylesheet in Listing 17-5 can perform this mapping.

Listing 17-5 MR Stylesheet—MR.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dp="http://www.datapower.com/extensions"
  extension-element-prefixes="dp" exclude-result-prefixes="dp"
  version="1.0">
  <xsl:output method="xml" />
  <xsl:template match="/">
    <!-- take the url and operation name from the ER output -->
    <xsl:variable name="url"
      select="resource/item[@type='original-url'] [1]/text()" />
    <xsl:variable name="opname"
      select="resource/item[@type='request-opname'] [1]/text()" />
    <xsl:variable name="resource"
      select="concat($url, '/', $opname, )" />
    <resource>
      <item type="custom">
        <xsl:value-of select="$resource" />
      </item>
    </resource>
  </xsl:template>
</xsl:stylesheet>
```

The ‘request-opname’ entry used is the result of selecting Local Name of Request Element in the resource extraction stage. This simple stylesheet concatenates the name of the operation to the URL that the client requested. Now we have a way of expressing the constraint in the external authorization system, by protecting the fictional URLs:

```
/bluebank/richpeople/currentaccount/getAmount
/bluebank/richpeople/currentaccount/addMoney
/bluebank/richpeople/currentaccount/removeMoney
```

Each of these “URLs” can then be given different authorization constraints in the external authorization system, and the AZ step acts as though the client requested the modified URL.

AZ—Authorization

The authorization step receives a large structure as its input. This node tree contains the output of the EI, MC, and MR steps, such that AZ can make a decision based on whether a given authenticated identity (EI and MC) can access a given resource (MR).

Many out-of-the-box authorization options already exist, and are often a perfect match to customer requirements for authorization. Sometimes, however, there is a need to go beyond these out-of-the-box options, either for integration with a custom external authorization system other than those directly supported, or to expand upon the built-in capabilities. An example of the latter is presented here.

Out-of-the-box integration with LDAP for authorization is implemented as one of the most common use cases—checking for membership of a specific LDAP group. This works by searching the LDAP server for the list of members in that group, and then comparing the output of the MC stage to that list. If the mapped credential is found, authorization succeeds; if not, then it fails. But what if we want to do more than checking for a single group membership? For instance, we might have a number of possible URLs (or Web service methods as demonstrated under MR earlier) and we would like to authorize different groups for each of the URLs, but without having to call out to an external authorization system. This may be an acceptable solution for a small number of static URLs, and would probably be easy to migrate later to a more robust enterprise authorization solution such as TAM.

One way that we might express such a set of authorization constraints would be to create a file that contains an XML nodeset, where the constraints are stated in a way that they can easily be retrieved using XPath. Listing 17-6 shows such a file.

Listing 17-6 One Way in Which Authorization Constraints Could Be Expressed

```
<?xml version="1.0" encoding="UTF-8"?>
<RESOURCES>
  <RESOURCE>
    <URL>/bluebank/richpeople/currentaccount/getAmount</URL>
    <GROUP>cn=workers,ou=groups,ou=example.com</GROUP>
  </RESOURCE>
  <RESOURCE>
    <URL>/bluebank/richpeople/currentaccount/addMoney</URL>
```



```

    <URL>/bluebank/richpeople/currentaccount/removeMoney</URL>
    <GROUP>cn=managers,ou=groups,ou=example.com</GROUP>
  </RESOURCE>
</RESOURCES>
</xsl:stylesheet>

```

This file would be written to the local file system (in this case to the file named `local:///AZresources.xml`) and will be called from the stylesheet; it could also be loaded from a remote server.

To process these constraints, our AU stylesheet must take the inputs from the MR step and know which resource the user is trying to access, take the input from the MC step and discover what groups the user is in, and then compare them to the expressed authorization constraints and make an authorization decision. The input to the AU stylesheet is shown in Figure 17-16.



Figure 17-16 The information from previous steps is available for authorization decisions.

To further complicate matters, real world LDAP structures are rarely simple and can sometimes require quite complicated interactions! The example in Listing 17-7 demonstrates an example of the kind of interaction with LDAP that is sometimes needed to determine group memberships and uses the authorization constraints stated in Listing 17-6 to make an authorization decision.

Listing 17-7 Custom Authorization Example

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dp="http://www.datapower.com/extensions"
  extension-element-prefixes="dp" exclude-result-prefixes="dp"
  version="1.0">
  <xsl:output method="xml" />
  <xsl:template match="/">

```

```

<xsl:variable name="perms"
  select="document('local:///AZresources.xml')"/>
<xsl:variable name="resource"
  select="/container/mapped-resource/resource/item/text()"/>
<xsl:variable name="DN" select="/container/mapped-credentials/
  entry[@type='custom']/authenticated/
  entry[@type='ldap']/text()"/>
<xsl:variable name="groups" select="dp:ldap-search(
  'ldapservers', '389', '', '$DN, 'ibm-allGroups', '',
  'BASE', '', '')"/>
<xsl:for-each select="$groups/LDAP-search-results/
  result/attribute-value[@name='ibm-allGroups']">
  <xsl:variable name="group" select="."/>
  <xsl:for-each select="$perms/RESOURCES/
    RESOURCE[URL=$resource]/GROUP">
    <xsl:if test=".= $group"><approved/></xsl:if>
  </xsl:for-each>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

The stylesheet makes an LDAP call to search for a specific attribute of the authenticated user. In this case, we use the IBM Directory Server, and that attribute, `ibm-allGroups`, is a special attribute of the directory server in use. The attribute returns a list of all the groups that a particular entity is a member of. We then go through the list of groups and search for one that matches any of the group entries in the permissions file for the requested resource. If we find a match, an XML node `<approved/>` is emitted and authorization succeeds; if not, no output is emitted, which results in an authorization failure. Indeed, any node set apart from `<approved>` will also be treated as authorization failure; the DataPower runtime is expecting to see an `<approved>` node.

TIP—LDAP OPTIMIZATION

While not part of the LDAP specification itself, many LDAP servers have a “special” attribute that returns a list of the groups for a given user. For the IBM Directory Server, this is `ibm-allGroups`, for Active Directory, it is `memberOf`, and so on. Using this attribute requests that the directory server itself calculate which groups the user is a member of; this is often significantly faster than going through a list of all of the members of a group and comparing them locally.

This is because in a typical directory server a given user will be a member of some relatively small number of groups; however, a group may have a very large number of members, so enumerating all of the members for comparison is much more expensive in terms of computation and network bandwidth than asking the directory server to do it for us.

PP—Post Processing

Finally, the Post Processing stage comes at the end and enables you to do literally anything based on the output of all the other stages. The post processing input includes

- Identity
- Credentials
- Mapped-credentials
- Resource
- Mapped-resource
- The original message

The default Post Processing options cover many scenarios. DataPower can generate an LTPA token, which is discussed in Chapter 21, “Security Integration with WebSphere Application Server.” It can generate an SAML assertion, a WS-Security UserNameToken, a WS-Security Kerberos token, or a SPNEGO token. It can call out to Tivoli Federated Identity Manager for token mapping. It can run some of the built-in post processing stylesheets such as `strip-wssec-header.xsl` to strip off WS-Security headers. Or it can execute a custom PP stylesheet to perform any action you want.

TIP—CREDENTIAL TRIMMING

When DataPower is processing incoming credentials and enforcing your security policies by performing authentication, authorization, and audit, it is likely that the credentials will not be required by the systems DataPower is proxying. If that is the case, they should be removed from the request before passing it on. This is best done in the PP stage with a simple custom stylesheet.

Working with Tivoli Security

Tivoli’s Access Manager (TAM) and Federated Identity Manager products (TFIM) are the IBM solutions for externalized authentication and authorization (TAM) and federated identity management (TFIM). These powerful products enable enterprisewide definition of policies for who is allowed to do what in the enterprise, and those policies can be enforced at policy enforcement points, such as DataPower.

Because the focus of this book is DataPower and not the Tivoli product set, the discussion in this section builds on existing work by the Tivoli DataPower integration team published in the TFIM documentation.

That documentation shows how to configure integration with the TAM and TFIM products; this chapter explores the configuration of TAM integration.

Integration with TAM

In order for a DataPower appliance to integrate with TAM, it must have two things: a TAM license on the device and the correct TAM libraries installed in the currently running version of firmware. The firmware contains a built-in configurable TAM client, able to communicate with and externalize authentication and authorization to a specific version of the TAM server. The place to check this is under Status→System→Device Features, where the feature TAM should appear and be set to Enabled, and under Status→System→Library Information, where the library TAM should appear with the correct version. At time of writing, supported TAM versions are version 5 and version 6, and you should use the version of firmware that contains the relevant client version for your TAM server.

Initial Configuration

Before any security integration with TAM can take place, the TAM client must be configured. The configuration process on DataPower works by completing a form to create some configuration files required for TAM integration. This can be found on the TAM configuration object page, which is under Objects→Access→IBM Tivoli Access Manager, as shown in Figure 17-17.

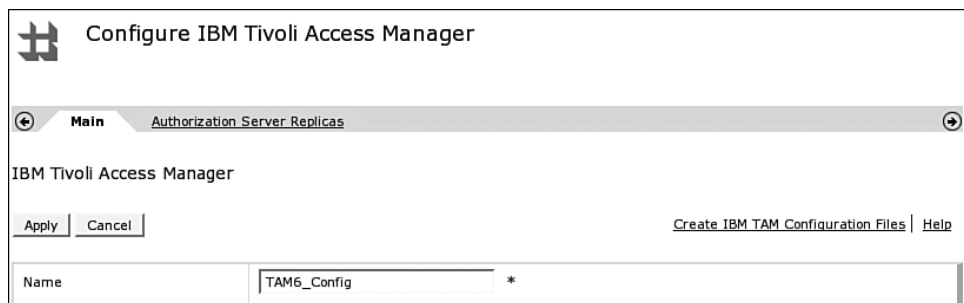


Figure 17-17 From the TAM Object Page, there is a link to create TAM configuration.

Clicking the Create IBM TAM Configuration Files link opens up a new window that determines how the files are to be generated. The window is shown in Figure 17-18. Note that in this example we have used static hosts defined on the DataPower device, for the hostname of the TAM Policy Server and of the LDAP Server, although this may use DNS.

After we click the Create IBM Tivoli Access Manager Configuration button shown in Figure 17-18, the appliance begins the file creation process. It makes a call out to the TAM policy server, authenticates to it using the credentials you have supplied, and asks the Policy server to register it as a valid server for sending authentication and authorization requests. As part of this process, TAM creates a new account for the DataPower appliance that it uses to bind to the LDAP server to perform queries as required. It provides the TAM CA certificate to the DataPower appliance such that SSL connections can be authenticated and trusted. Finally all the TAM configuration files are saved to the relevant areas on the DataPower file system. The DataPower logs of this process are shown in Figure 17-19.

Create IBM Tivoli Access Manager Configuration

Create File Copies to Download

on

off

Output Configuration File Name

TAM6_config

*

Tivoli Access Manager Administrator

sec_master

*

Administrator Password

*

Tivoli Access Manager Domain

Default

*

Tivoli Access Manager Application ID

datapower1

*

Policy Server

TAM-POLICY-SERVER

*

Policy Server Manager Port

7135

*

SSL Key File Lifetime

183

Days

*

SSL Timeout

30

Seconds

*

LDAP Server

TAM-LDAP-SERVER

*

LDAP Server Port

389

*

LDAP Administrator Password

*

LDAP Authentication Timeout

30

Seconds

*

LDAP Search Timeout

30

Seconds

*

Enable LDAP Cache

on

off

Create IBM Tivoli Access Manager Configuration

Figure 17-18 TAM configuration.

msgid	message	Show last 50 100 all
0x81a0001f	Created Tivoli Access Manager LDAP user registry configuration file	
0x81a00031	Starting LDAP registry file creation	
0x81a0001f	Created Tivoli Access Manager host runtime configuration file	
0x81a0001f	Created Tivoli Access Manager key database and key stash configuration file	
0x81a00028	Generating Tivoli Access Manager host config succeeded	
0x81a00020	Tivoli Access Manager server CA certificate was successfully retrieved as part of client configuration	
0x81a00026	Starting Tivoli Access Manager file creation process	
0x81a0000d	Signaling Tivoli Access Manager client file creation	

Figure 17-19 TAM configuration logs.

The TAM configuration files that are created consist of four separate files:

1. A text file with a suffix of .conf, which contains specific information obtained from the TAM server about the TAM configuration.
2. A binary file with a suffix of .conf.obf, which contains the same information as the .conf file but in an obfuscated file, and can be used instead of the conf file if there is a need for the passwords used to connect to the LDAP server and TAM to not be stored in plain text.
3. A key database file with a suffix of .kdb, which contains the SSL certificates used to validate the SSL connection with the TAM policy server.
4. A stash file with a suffix of .sth, which contains the stashed password to the key database file.

The names of the files are determined by the Output Configuration File Name option in the configuration screen; in our example from Figure 17-18, the files would be called TAM6_config.conf, TAM6_config.kdb, and so on. Note that the key database file and related stash file are, by default, created only in the `cert://` file system, from which they cannot be exported. If you have a need to export these files (for instance for backup purposes), the Create File Copies to Download option should be selected when creating the configuration; they will then be copied to the `temporary://` file system from which they can be downloaded. The other two files are stored to the `local://` file system of the domain.

Application ID

One of the fields shown in Figure 17-18 is called the “Tivoli Access Manager Application ID.” This field is used to create a server definition for this specific DataPower appliance in the TAM database and a server account in the LDAP server used as its repository.

The server definition and account that is created uses both the application ID and the System ID of the DataPower appliance. The System Identity of the DataPower device is configured at an appliance level in the Default domain, by navigating to Administration→Device→System Settings and filling in the System Identifier field. Crucially, whatever is put in here must be resolvable on the TAM server, and must resolve to the DataPower appliance, otherwise the configuration will fail. This can be done by modifying the hosts file on the TAM server, or by specifying a resolvable fully qualified DNS name as the System Identifier. In our example, we used the Application ID `datapower1` and a System ID of `DataPowerBookXI50` (and `DataPowerBookXI50` was added to the hosts file of the TAM server), thus the server definition is `datapower1-DataPowerBookXI50` and the account principal is `datapower1/DataPowerBookXI50`.

The Application ID is used in TAM for application-specific authorization purposes. Thus two DataPower devices that serve the same “application” (for instance which both proxy the same Web service and have a Load Balancer in front of them) should use the same Application ID. Their entries in the TAM namespace differ because of the different System ID’s with which they should be configured.

If you have a need to configure TAM multiple times on the same appliance, the application ID’s used *must* be different. For instance, you may want to have multiple domains all integrating with TAM for development or testing purposes. Because the System ID is a device-level setting configured in the Default domain, no two DataPower domains on the same device should ever use the same Application ID when connecting to the same TAM server.

SYSTEM IDENTIFIER

The System ID of the DataPower appliance must be set for a TAM configuration to work correctly. If you do not set this on your device, the TAM client will still connect and configure itself, however rather than using the System ID for the creation of the server definition and the account principal, it will use the value entered for the TAM policy server on the configuration screen. Things appear to work, but you have incorrect configuration on the TAM server, and this may cause serious problems later. It is strongly recommended that DataPower appliances using TAM should have correctly configured System IDs.

Configuration Failures

If the TAM configuration fails, this will be clear from the logs in the domain. The failure message likely contain a TAM error; these can be decoded by looking at the TAM documentation, or by asking a friendly TAM administrator. The most likely causes of TAM configuration failure are

- Lack of network connectivity. There must be network connectivity between DataPower and the TAM server, not just at runtime but also during configuration. As mentioned, the configuration process involves connecting to and registering with the TAM policy server, and if this is not possible, it will result in a failed configuration.
- Incorrect policy server port. Most TAM policy servers run on the default ports of 7135 and 7136, but this is configurable, and some TAM administrators may have chosen to use a different port.
- Incorrect username or password used for connection to the TAM server or the LDAP server.
- An account for the specific application ID for a given appliance with a given system identifier already exists in this TAM server, perhaps from a prior configuration attempt.
- Incompatible version of TAM client. For instance using a TAM5 DataPower firmware will not work with a TAM6 Server if that TAM6 server is configured in FIPS mode.

Any issues with the TAM configuration must be resolved before continuing. Do not go on to creation of TAM configuration objects until the TAM configuration has completed successfully and been verified in the logs.

Unconfiguration

If the configuration partially succeeds, or if you need to remove an existing configuration, it is not as simple as just removing the files that are created on the DataPower appliance. Because the configuration process connects to the Access Manager server and makes configuration changes, those must also be removed. This is especially important if you intend to use the same application identity, because the configuration will simply fail.

The DataPower TAM client does not include a utility to remove the TAM configuration. Thus removing the entries from the TAM database and LDAP server must be done on the server itself. TAM provides a command, `svrsslcfg`, which is used to manually add and remove server configuration from the database and LDAP server. This command requires a configuration file, an application ID, and a hostname. For removal, it is possible to simply use an empty configuration file; the application id (-n) and the hostname (-h) are the parameters that decide the server definition to be removed. The `touch` command can be used to create an empty configuration file, which is then passed to the `svrsslcfg` command. This process is shown in Listing 17-8.

Listing 17-8 Unconfiguring a DataPower Appliance from TAM

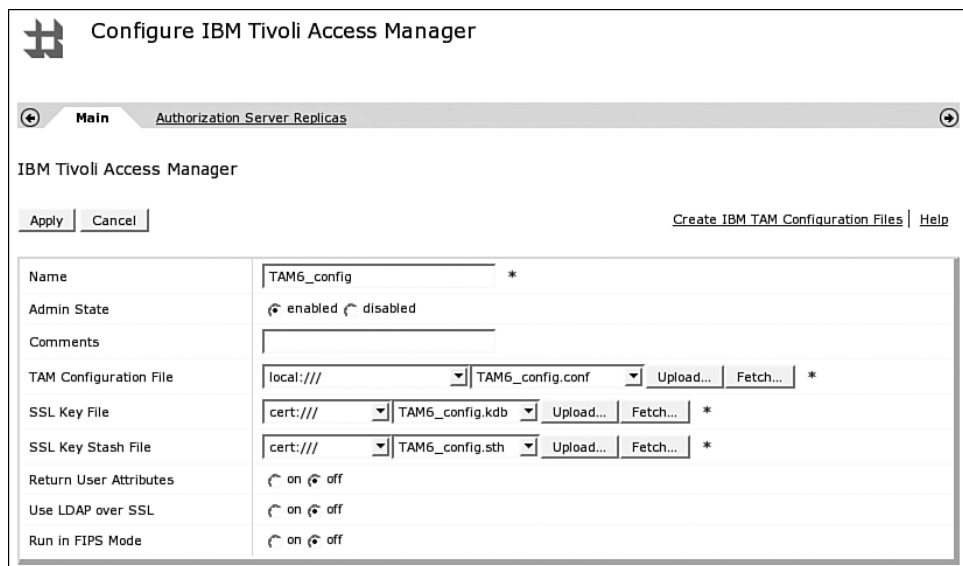
```

tam-server:~ # pdadmin -a sec_master
Enter Password: *****
pdadmin sec_master> server list
    i vacld-tam-server
    datapower1-DataPowerBookXI50
pdadmin sec_master> quit
tam-server:~ # touch dummy.conf
tam-server:~ # svrsslcfg -unconfig -f dummy.conf -n datapower1 -h
DataPowerBookXI50
Enter the password for sec_master:
*****
Unconfiguration of application "datapower1" for host "DataPowerBookXI50" is
in progress.
This might take several minutes.
SSL unconfiguration for application "datapower1" has completed
successfully.
tam-server:~ # pdadmin -a sec_master
Enter Password:
pdadmin sec_master> server list
    i vacld-tam-server
pdadmin sec_master> quit
tam-server:~ #

```

Object Creation

After the four configuration files are successfully created, we can go ahead and create a TAM configuration object referencing those files, as shown in Figure 17-20.



Configure IBM Tivoli Access Manager

Main Authorization Server Replicas

IBM Tivoli Access Manager

Apply Cancel Create IBM TAM Configuration Files Help

Name	TAM6_config *
Admin State	<input checked="" type="radio"/> enabled <input type="radio"/> disabled
Comments	
TAM Configuration File	local:/// TAM6_config.conf Upload... Fetch... *
SSL Key File	cert:/// TAM6_config.kdb Upload... Fetch... *
SSL Key Stash File	cert:/// TAM6_config.sth Upload... Fetch... *
Return User Attributes	<input type="radio"/> on <input checked="" type="radio"/> off
Use LDAP over SSL	<input type="radio"/> on <input checked="" type="radio"/> off
Run in FIPS Mode	<input type="radio"/> on <input checked="" type="radio"/> off

Figure 17-20 TAM configuration.

One of the most important fields on Figure 17-20 is the Run in FIPS Mode field. The Federal Information Processing Standard (FIPS) standard specifies explicitly the types of encryption that can be used. If you are working with a TAM server that uses FIPS-compliant encryption, you will be able to communicate with it *only* if this button is selected. FIPS mode is configured in the SSL stanza of the TAM policy server's `pd.conf` file—if there is an entry like `ssl-enable-fips = yes` then the Run in FIPS Mode button must be set to on, otherwise you will see in the logs the error shown in Figure 17-21.

```
tam (TAM6_config): Tivoli Access Manager authorization client
log message: TAM server error: azn_initialize : HPDBA0234E
The SSL communications could not be completed. The socket
was closed.
```

Figure 17-21 Incorrect FIPS settings will result in an SSL error.

The second tab of the object screen also needs to be configured with at least one Authorization server replica. (Of course, if there are more we could specify them all here.) In many instances the main authorization server replica resides on the policy server, as it does for our example environment, so we have used the same host alias as before in Figure 17-22. (Note that this is a different port to the manager port shown in Figure 17-18—the default authorization port is 7136, and specifying the wrong port here can be very hard to debug!)

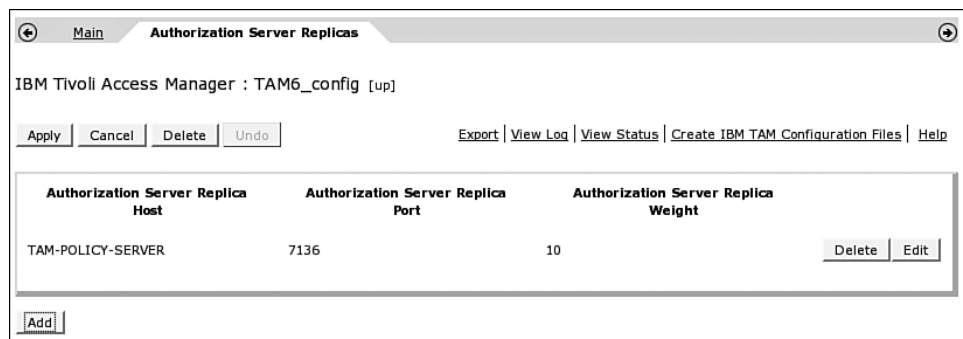


Figure 17-22 At least one Authorization Server replica is required.

After clicking Apply on the TAM object configuration screen, we see the object initially go into a red Pending state, shown in Figure 17-23.

Name	Status	Op-State	Logs	Admin State	TAM Configuration File	SSL Key File	SSL Key Stash File
TAM6_config	new	Pending		enabled	local:///TAM6_config.conf	cert:///TAM6_config.kdb	cert:///TAM6_config.sth

Figure 17-23 Initial Pending state.

This is not a cause for alarm—this is simply the state it goes into when the initial configuration is being verified, because it takes a couple of seconds for everything to fall into place. The logs are shown in Figure 17-24.

08:45:09	mgmt	notice	59088065		0x00350014	tam (TAM6_config): Operational state up
08:45:09	tam	notice			0x81a0001d	tam (TAM6_config): Tivoli Access Manager client successfully enabled
08:45:07	tam	info			0x81a00038	tam (TAM6_config): The Tivoli Access Manager object has been configured; attempting to start client authorization endpoint
08:45:07	mgmt	info	59088065		0x00360001	tam (TAM6_config): Pending
08:45:07	mgmt	notice	59088065		0x00350015	tam (TAM6_config): Operational state down
08:45:07	tam	info	59088065			tam (TAM6_config): starting tam thread
08:45:07	tam	debug	59088065		0x81a0003b	tam (TAM6_config): Successfully added replica to Tivoli Access Manager configuration - hostname 'TAM-POLICY-SERVER', port 7136, weight 10
08:45:06	tam	debug	59088065		0x81a00036	tam (TAM6_config): 1 authorization replica configured
08:45:06	tam	debug	59088065		0x81a00033	tam (TAM6_config): Created LDAP registry file

Figure 17-24 A successful configuration.

The object creation is now complete and can be used in AAA processing policies.

AAA Processing

Once the TAM configuration is complete and the TAM objects exist, configuring for AAA authentication is as easy as selecting the option and pointing it at the TAM configuration. Figure 17-25 shows the configuration for AAA authentication.

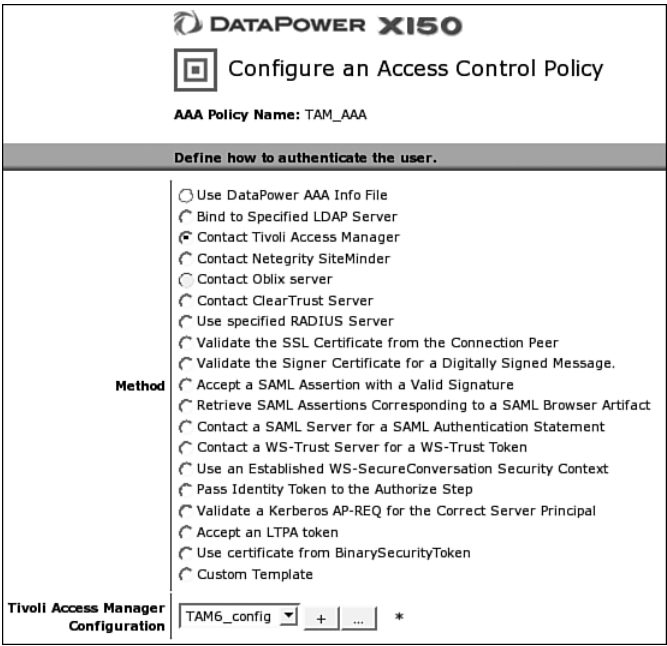


Figure 17-25 AAA Authentication with TAM.

Authorization, however, is rather more involved. TAM authorization has many aspects, and permissions can be granted at many different levels. Because the TAM runtime is very flexible, there are many possible ways that access control can be configured, and it is beyond the scope of this book to examine them in detail. Instead we will show a specific example of configuration to demonstrate the flexibility and power of the combination of TAM and DataPower.

The AZ step to communicate with TAM takes three inputs, which can be seen in Figure 17-26, which displays the Probe extension trace of the input into a TAM AZ step.

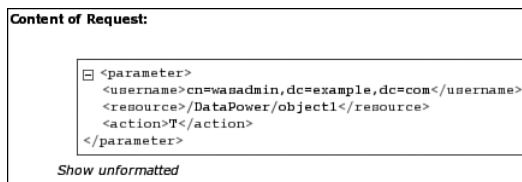


Figure 17-26 Inputs to the TAM AZ step.

The inputs are the authenticated username (which in this instance came directly from a TAM AU step, but could of course have come from a custom AU step or an MC step), the resource being requested (which in this case came from the URL sent by the client, but could of course have come from any other ER or MR step), and the action that DataPower believes the client is trying to perform. The resource and the action here require more explanation.

First, we will examine the resource. In order to understand how TAM makes authorization decisions, we have to look briefly at the TAM object space. This object space is just a hierarchical listing of objects that can have access control lists (ACL) attached to them. When TAM receives a request for authorization of a resource, that resource is mapped directly to the object space. To demonstrate this, we set up an object space with a very simple resource structure. At the top level is the object /DataPower, and underneath it are three separate objects: object1, object2, and object3. We have chosen this object space to be a simple representation such that we can easily map it to URLs, which is protected by the object space. The URLs to be protected are shown in Listing 17-9.

Listing 17-9 The URL Structure to Protect

```
http://datapower-ip-address/DataPower/object1
http://datapower-ip-address/DataPower/object2
http://datapower-ip-address/DataPower/object3
```

Listing 17-10 shows the object space of these specific objects in our TAM server and a description of one of the objects.

Listing 17-10 TAM Object Space Configured to Protect Our Simple URL Structure

```
pdadmin sec_master> object list /DataPower
/DataPower/object1
/DataPower/object2
/DataPower/object3
pdadmin sec_master> object show /DataPower/object1
Name: /DataPower/object1
Description: Test object 1
Type: 14 (Application Container Object)
Is Policy Attachable: Yes
Extended Attributes:
Attached ACL: DataPower-acl1
Attached POP:
Attached AuthzRule:

Effective Extended Attributes:
Effective ACL: DataPower-acl1
Effective POP:
Effective AuthzRule:
pdadmin sec_master>
```


The objects in the TAM namespace have an attached Access Control List (ACL). This is similar to, but far more advanced than, a UNIX[®] file permission. The ACL attached to our objects is shown in Listing 17-11.

Listing 17-11 The ACL Attached to Our Objects

```
pdadmin sec_master> acl show DataPower-acl1
ACL Name: DataPower-acl1
Description:
Entries:
Any-other T
User wasadmin Tr
pdadmin sec_master>
```

This ACL is the list of permissions that a specific user has on the object. In this case all authenticated users have the T permission, and the user called wasadmin has both the T and r permissions. T stands for Traverse, while r stands for read. The T permission is required to Traverse the tree; if a user does not have Traverse permission on the object `/DataPower`, they cannot access anything under it such as `/DataPower/object1`. The r permission seems like it should mean that the user can read the resource. However, what that actually means is specific to the application that is mapping the resource.

Back in Figure 17-23, we showed the input into the TAM AZ step, which included an authenticated user, a resource, and an action. The action in Figure 17-23 was T; that is, DataPower was asking TAM “Does this user have the T permission on this resource?” The reason it asked about T permission was because this is how the default AZ action is configured. Figure 17-27 shows the TAM AZ configuration.

 **Configure AAA Policy**

[Authenticate](#) | [MapCredentials](#) | [Resource](#) | [MapResource](#) | **Authorize** | [Pos](#)

AAA Policy : TAM_AAA [up]


[Apply](#) | [Cancel](#) | [Delete](#) | [Undo](#)


Method	Contact Tivoli Access Manager *
Cache authorization results	Absolute *
Cache Lifetime	3 Seconds
Tivoli Access Manager Configuration	TAM6_config + ... *
Default Action	T (Traverse)
Resource/Action Map	+ ...

Figure 17-27 The TAM AZ step.

The Default action field defines what permission bit we will be asking TAM for when making the authorization call. Because it is set to T, anyone with T permission on the object that matches the resource will be successfully authorized. Our ACL stated that all users have T permission on the object, so anyone who authenticates can access this object. If however we change this to r, only the user named wasadmin, as the only one with r permission listed in the ACL, will be able to access the object.

Finally, we can create a construct called a Resource/Action map, which allows you to dynamically map an operation such as a URL to a required permission bit based on a simple matching, as shown in Figure 17-28.

 [Help](#)

 **Edit TAM AAA Authorization Resource/Action Map**

TAM authorization Resource/Action mappings

Resource Pattern	TAM Action	WebSEAL Conformance	
getAmount	r (Read)		^ v
addMoney	x (Execute)		^ v

[Add](#) | [Cancel](#) | [Back](#) | [Next](#)

Figure 17-28 The Resource/Action map.

Summary

This chapter has provided an in-depth look at customizing the AAA process, which is extremely flexible and powerful and enables integration with any form of AAA processing imaginable. We have also gone through externalizing of authentication and authorization to IBM's Tivoli Access Manager, the IBM solution to Enterprise Class centralized Access Control. And yet this knowledge has barely scraped the surface of what is possible with the AAA runtime; you can now go and explore to create the AAA processing that suits your specific requirements and needs!