1 External User Authentication

1.1 Goals:

- Add Aruba's unique value (security, low-latency mobility, wireless IDS/IDP, policy based redirection) to deployments using external authentication servers such as standalone captive portal solutions for "advanced hotspots"
- Leverage capabilities of existing external captive portal solutions (credit card billing, alternative authentication such as iPass or Boingo, hospitality system integration, backend billing system integration, etc.)
- Provide for an alternative method of authentication and remediation for a class of users, while simultaneously allowing other users to use existing authentication methods. For example, Guest Users require captive portal authentication with appropriate billing while employees have secure access to internal resources and at the same time, voice devices are limited to call setup and voice traffic only.
- Enable load balancing of captive portal appliances for more centralized or large scale deployments
- Extend to wireless and wired access control
- Provide an extendable method for implementing external authentication and remediation solutions in a network. Unauthenticated users are provided with limited access to an authentication/remediation server and then with information from the client, the server can then decide the policies for the switch to enforce for that user.

1.2 Deployments:

- Hotspots- add security, voice QoS and other services while leveraging existing infrastructure
- Hotels- guests, hotel staff, voice, special events/groups
- Convention centers- guests, hotel staff, events, voice
- "Leased" hotspots- hotspot AP(s) deployed inside corporate area
- Third Party Remediation – Third party client and server software can be made to interact with the network in order to limit network access to the client without proper credentials.

1.3 Implementation assumptions:

- Standards-based interface (XML, HTTP/HTTPS, RADIUS)
- Secure Interface (HTTPS)
- Extensible. The ability to add further information elements in the future
- Named policies defined in the switch such as role names must be manually configured in the external XML client. There is no mechanism for this information to be learnt.

1.4 Interfaces Formats:

- HTTPS GET with elements encoded in URL
- XML (in HTTPS POST)

1.5 HTTPS GET

The HTTPS GET method of creating users on the switch leverages the existing internal captive portal authentication mechanisms. Once a user has been redirected to a web site, this web site must instruct the user's browser to generate a https get request which sends the username and password to the switch for authentication.

This method of user creation defines only 2 variables:
"user" the name of the user.
"password" the users password.

NOTE: "username" is treated as an alias for "user"

An example URL that the user must request is given below:
https://<switch-ip>/auth/loginnw.html?user=foobar&password=foobar

NOTE: it is also possible to use the following URL:
https://<switch-ip>/auth/loginnw.html?username=foobar&password=foobar

Once a client requests one of the above URLs, the switch will automatically add a user into its user table with the IP address equal to the source IP of the https request and authenticate the user using the configured authentication mechanism. The role of the user is derived as normal, using the aaa derivation rules, or without any derivation rules, the default role is chosen.

If the authentication is unsuccessful then the switch will reset the role of that user to be "logon" and update the user name with the information from the request.

The key point with this method is that the user can only authenticate themselves since the IP address of the user installed in the switch is the source IP address of the request. The https protocol ensures that the user's source address is not spoofed.

2 XMLInterface

The XML interface defines a method for the Aruba switch to accept notifications from an external server that the role of a user should be updated. The external server is expected to know the predefined roles in the switch and using some external method of user authentication or remediation, notify the switch of a new role for the user when appropriate.

The interaction between the external server and the switch is performed as a result of the external server issuing commands via a https POST to the switch and the switch responding with the result of the command issued.

NOTE: This document defines the XML-API protocol interface v1.0

The external server must use the following URL in order to issue commands to the switch:
https://<aruba-switch-ip>/auth/command.xml

2.1 Switch Configuration:

The administrator of the switch must provision the IP address of the external server into the switch using the following CLI syntax:

(switch) (config) # **aaa xml-api client A.B.C.D**

Where A.B.C.D is the IP address of the external server.

NOTE: switch command enters a configuration sub-mode where the following command may be entered:

(switch) (ecp-client) key <Client_Key>

Where <Client_Key> is the shared secret that exists between the switch and the external captive portal server in order to ensure the identity of the external server.

2.2 Server XML:

The server will POST XML to the switch with a syntax form similar to the following:

<aruba command="COMMAND">
<tag1>Value1</tag1>
...
<tagn>Valuen</tagn>
</aruba>

2.2.1 Available commands:

user_add: Add a user into the switches user table
user_delete: Remove a user from the switches user table

user_authenticate: Authenticate a user using the switches configured authentication mechanism.
user_blacklist: Deny association requests for this user
user_query: Returns the current state of the user

2.2.2 Available tags

ipaddr: IP address of the user in A.B.C.D format
macaddr: MAC address of the user aa:bb:cc:dd:ee:ff format (with colons)
user: Name of the user. It is a string of maximum size 64
role: Role name is a string of maximum size 64
password: The password of the user to use when authenticating the user.
session_timeout: Session timeout in minutes. User will be disconnected after expiry of this time period.
authentication: Authentication method to authenticate the message and sender.

This is one of MD5, SHA-1 or clear text. This tag is ignored if shared secret is not configured and mandatory if configured key. This is encoded SHA1/MD5 hash of shared secret or plaintext shared secret. This tag is ignored if shared secret is not configured on the switch. Note that the actual MD5/SHA-1 hash is 16/20 bytes and consists of binary data. It must be encoded as an ASCII based HEX string before sending. It must be present when the switch is configured with an xml-api key for the server.
Encoded hash length will be 32/40 bytes for MD5/SHA-1.

version: Version is the version number. Currently this must be set to 1.0. This field is mandatory is all requests.

The following sections list the tags that may be used with each command. After each command, there is a list of the mandatory fields. These are the fields that must be present in order for the switch to perform the requested command. The fields that are not listed as mandatory will be used to update the user table of the switch if present. Tags other than those listed below will return an error.

NOTE: when the switch has been configured with a secret key, the Authentication and key fields are mandatory in all circumstances. They are however not listed in the sections below as being mandatory. Configuring a secret key on the switch is highly recommended.

2.3 Command XML

2.3.1 Adding a User: user_add

This command creates a new user entry if no user is currently defined, or updates an existing entry.

<aruba command=”user_add”>
<ipaddr>Ipaddr</ipaddr> ***Manditory Field***
<macaddr>Macaddr</macaddr>

<name>User_Name</name>
<role>Role_Name<role>
<session_timeout>Session_timeout</session_timeout>
<key>Key</key>
<authentication>MD5|SHA-1|cleartext</authentication>
<version>1.0</version> ***Manditory Field***
</aruba>

2.3.2 Delete User: user_delete

This command deletes an existing user entry. If multiple attributes are present they all must match before the entry will be deleted.

<aruba command="user_delete">
<ipaddr>Ipaddr</ipaddr> ***Manditory Field***
<macaddr>Macaddr</macaddr>
<name>User_Name</name>
<key>Key</key>
<authentication>MD5|SHA-1|cleartext</authentication>
<version>1.0</version> ***Manditory Field***
</aruba>

2.3.3 Authenticating a User: user_authenticate

This will cause the switch to send a PAP authentication request to authentication server in order to authentication and derive the role of the user.

<aruba command="user_authenticate">
<ipaddr>Ipaddr</ipaddr> ***Manditory Field***
<macaddr>Macaddr</macaddr>
<name>Name</name> ***Manditory Field***
<password>Password</password> ***Manditory Field***
<key>Key</key>
<authentication>MD5|SHA-1|cleartext</authentication>
<version>1.0</version> ***Manditory Field***
</aruba>

2.3.4 Blacklist User: user_blacklist

This will cause Aruba switch to blacklist the specified user.

<aruba command="user_blacklist">
<ipaddr>Ipaddr</ipaddr> ***Manditory Field***
<macaddr>Macaddr</macaddr>
<name>User_Name</name>
<key>Key</key>
<authentication>MD5|SHA-1|cleartext</authentication>
<version>1.0</version> ***Manditory Field***
</aruba>

2.3.5 Query user: user_query

This queries the Switch records and returns information about the user.

<aruba command="user_query">
<ipaddr>Ipaddr</ipaddr> ***Manditory Field***
<macaddr>Macaddr</macaddr>
<name>User_Name</name>
<key>Key</key>
<authentication>MD5|SHA-1|cleartext</authentication>
<version>1.0</version> ***Manditory Field***
</aruba>

2.4 XML response

Following sections define the format of the response sent in reply to XML commands:

2.4.1 Normal Response

This text is returned in response to all commands except the *user_query* command. It take the following form:

<aruba>
<result>Result</result>
<code>Code</code>
<reason>Reason</reason>
</aruba>

Result is either "Error" or "Ok"
Code is an integer number denoting specific error occurred in executing the command. It is sent only when result is "Error".
Reason is a message string describing error.

The following table shows all the result codes, the commands that can generate them and their corresponding reason messages as of this version of the API:

1. **unknown user** - user_authenticate, user_delete, user_blacklist, user_query
2. **unknown role** - user_add
3. **unknown external agent** - any command
4. **authentication failed** - any valid command but only when a key is configured on the switch
5. **invalid command** - when the command is not valid- refer to the "available_commands" section above
6. **invalid message authentication method** - any valid command, when a key is configured on the switch
7. **invalid message digest** - any valid command, when a key is configured on the switch
8. **missing message authentication** - any valid command, when a key is configured on the switch

2.4.2 Response to user_query command

The user_query command responds with the following style of text:

<aruba>
<result>Result</result>
<code>Code</code>
<reason>Reason</reason>
<role>Role</role>
<type>Type</type>
<auth_status>Auth_status</auth_status>
<auth_server>Auth_server</auth_server>
<auth_method>Auth_method</auth_method>
<location>Location</location>
<age>Age</age>
<essid>Essid</essid>
<bssid>Bssid</bssid>
<phy_type>Phy_type</phytype>
<vlan>Vlan</vlan>
</aruba>

Where
Result: Same as the normal response
Code: Same as the normal response
Reason: Same as the normal response

NOTE: the following tags are only present when the Result is "OK".
Role: Role of the user
Type: Either "wired" or "wireless"
Auth_status: Either "authenticated" or "unauthenticated"
Auth_server: Name of the authentication server used. Only present if the user 0has been authenticated by the switch.
Auth_method: The authentication method that was used to authenticate the user. Only present if the user has been authenticated by the switch.
Location: The current location of the user. For wireless clients, the AP location in B.F.L format is returned. For wired users the slot/port is returned
Age: Age of the user in DD:HH:MM format
ESSID: ESSID that user associated to.
BSSID: BSSID of the AP that the user is currently associated with.
Phy_type: One of "a", "b" or "g"
Vlan: The VLAN that the user currently has

2.5 Monitoring External Clients from the Switch

The switch provides CLI commands to check the status of the external XML API clients.

2.5.1 Displaying the Configuration

The "show aaa xml-api clients" command displays all the configured xml API clients on the switch along with the shared secret. By default all secrets on the switch are not displayed. In order to check the real shared secret, the configuration mode command "encrypt disable" can be used to display the secrets.

```
(switch) # show aaa xml-api clients
ECP Client Configuration
------------------------
IP Key
----------- ---
10.215.1.3 *****
10.215.250.240 *****
(switch) #
```

2.5.2 Displaying Statistics

The "show aaa xml-api statistics" command outputs the number of times that the particular event occurred per client. Each client has 2 numbers associated with it, the total number of times that this event has occurred is displayed first, with the number of new events since the last time the counters were displayed is shown in parentheses "("and ")".

```
(switch) (config) # show aaa xml-api statistics
ECP Statistics
--------------
Statistics 10.3.8.253
---------- ----------
user_authenticate 1 (0)
user_add 1 (0)
user_delete 1 (0)
user_blacklist 2 (0)
unknown user 2 (0)
unknown role 0 (0)
unknown external agent 0 (0)
authentication failed 0 (0)
invalid command 0 (0)
invalid message authentication method 0 (0)
invalid message digest 0 (0)
Packets received from unknown clients : 0 (0)
Packets received with unknown request : 0 (0)
Requests Received/Success/Failed : 5/3/2 (0/0/0)
```

2.6 XML DTD

```
<!-- DTD for External Captive Portal Request-->
<!ELEMENT aruba - -
(ipaddr | macaddr | name | role | session_timeout | authentication |
key | version)*
```

```
-- global envelope -->
<!ATTLIST aruba
command (user_add|user_delete|user_authenticate)
#REQUIRED -- commands
<!ELEMENT ipaddr - - (#PCDATA) -- IP address -->
<!ELEMENT macaddr - - (#PCDATA) -- MAC address -->
<!ELEMENT name - - (#PCDATA) -- Name -->
<!ELEMENT role - - (#PCDATA) -- Role Name -->
<!ELEMENT session_timeout - - (#PCDATA) -- Session Timeout -->
<!ELEMENT password - - (#PCDATA) -- Cleartext password -->
<!ELEMENT authentication - - (#PCDATA) -- Authentication Method -->
<!ELEMENT key - - (#PCDATA) -- Hashed shared secret -->
<!ELEMENT version - - (#PCDATA) – Version number -->
<!-- EOF -->
<!-- DTD for External Captive Portal Result -->
<!ELEMENT aruba - -
(code | reason | result)*
-- global envelope -->
<!ATTLIST aruba
result (Error | Ok) # REQUIRED -- result code
<!ELEMENT Code - - (#PCDATA) -- Error Number -->
<!ELEMENT Reason - - (#PCDATA) -- Error message string -->
<!-- EOF -->
```

3 Sample Applications:

3.1 External Captive Portal

Call Flow:
Using RADIUS authentication:

a. The user associates to AP/switch and is redirected to Captive Portal Server.
b. The user authenticates to Captive Portal after providing sufficient payment such as a credit card.
c. The Captive Portal server updates the RADIUS server with the correct information such as Username, Password and Session Timeout in order to allow the client to authenticate to the switch.
d. The Captive Portal server POSTs the user information to the switch using the XML interface's “user_authenticate" command, or alternatively uses http redirection to force the client to GET the correct username/password URL from the switch.
e. The Aruba switch authenticates the user against it's configured RADIUS server. Since the captive portal server updated the RADIUS server with the user's login information and appropriate attributes, the switch uses the RADIUS response to derive the apropriate user profile such as role, session timeout and more.
f. Once the user's timeout has occurred, the user will be returned to logon role and will automatically be redirected to the external captive portal for further credit payment. Alternatively, the user logs out, or disconnects from the switch and the switch updates the RADIUS server with the logout information.

Not using RADIUS authentication (e.g. iPass, Boingo, internal user DB:

a. The user associates to AP/switch and is redirected to Captive Portal Server
b. The user authenticates to Captive Portal after providing sufficient payment such as a credit card
c. The Captive Portal server POSTs the user information to the switch using the XML interface's “user_add” command. This automatically updates the user's role and instantly provides the user with the requested profile.
d. Once the user's timeout has occurred, the user will be returned to logon role and will automatically be redirected to the external captive portal for further credit payment.

3.2 Differentiated access for guests and staff.

Guest users are forced to authenticate themselves using captive portal by having their traffic redirected to the external captive portal. The external captive portal server can issue the clients with a onetime password for the switch to authenticate against, or it can simply update the role of the switch directly.
Staff members are authenticated through other methods, such as 802.1x or VPN. When using a different authentication method, the switch can automatically update the role of

the user to bypass captive portal but still provide appropriate access control, QoS, VLAN derivation and bandwidth control across all users.

3.3 Leased hotspot

With a leased hotspot service, an infrastructure operator can deploy their access points in hotspot areas and then wholesale the wireless access to multiple hotspot operators. By utilizing Aruba derivation rules, clients that access specific ESSIDs can be provided with different styles of logon role. Each logon type role will be defined with limited access and a captive portal style session acl that redirects the users' http requests to the captive portal server associated with that ESSID.

The switch configuration can be implemented like this:

```
(sw) (config)# ip access-list session cp-ssid1
(sw) (config-sess-cp-ssid1)# user alias A.B.C.D svc-https permit
(sw) (config-sess-cp-ssid1)# user any svc-http dst-nat ip A.B.C.D Port#
(sw) (config-sess-cp-ssid1)# user any svc-https dst-nat A.B.C.D sPort#
(sw) (config-sess-cp-ssid1)# exit
(sw) (config)# user-role logon-ssid1
(sw) (config-role)# session-acl control
(sw) (config-role)# session-acl cp-ssid1
(sw) (config-role)# exit
(sw) (config)# aaa derivation rules user
(sw) (user-rule)# set role condition essid equals ssid1 set-value
logon-ssid1
```

With this setting, a user that connects to the SSID of ssid1 will automatically derive the user role of “logon-ssid1”. This role has been defined to force all users to be redirected to the external captive portal at address A.B.C.D with a TCP port of Port# for normal http traffic and sPort# for users that try to connect with https.

4 Sample Source Code

Sample C code is available on request. The following sections document sample code to generate and post the required XML text using the perl language.

Prior to using these scripts, you will need to ensure that you have the URI, LWP and Digest::MD5 and Digest::SHA1 modules correctly installed in your machine.

For detailed instructions, please reference your perl manuals, however with a correctly installed cpan application, you should be able to install the modules with the following commands:

```
root # cpan
cpan shell -- CPAN exploration and modules installation
(v1.7601)
ReadLine support available (try 'install Bundle::CPAN')
cpan> install URI
...
cpan> install LWP
...
cpan> install Digest::MD5
...
cpan> install Digest::SHA1
aruba_xml.pl:
#!/usr/bin/perl -w
use strict;
##############################
#
# This is the sample perl software for sending
# XML Posts to the Aruba AirOs Switch
#
# Copyright (C) 2002-2005 by Aruba Wireless Networks, Inc.
# All Rights Reserved.
#
# A Sample XML post looks like this:
#
# ----8<----8<----8<----8<----8<----8<----
# xml=<aruba command=user_add>
# <ipaddr>1.2.3.4</ipaddr>
# <name>Mr_Blibble</name>
# <role>logon</role>
# <authentication>cleartext</authentication>
# <key>supersecret</key>
# </aruba>
# ----8<----8<----8<----8<----8<----8<----
#
# To update this code to support new commands, simply
# add the new command into @ARUBA_VALID_COMMANDS
#
# To update this code to support new user attributes,
# simply add the attribute into @ARUBA_VALID_USER_ATTRIBUTES
#
# To update this code to support new authentication methods,
# add the new auth method into @ARUBA_VALID_AUTH_METHOD and
# also add support in aruba_generate_xml in order to
```

```
# generate the correct key
#
#
########################################
################
# First we must include the required libraries.
#
# Be sure to install these into your system. You can find them
# by searching at:
# http://search.cpan.org/search?module=<module_name>
#
# For example, URI::Escape is found by going to:
# http://search.cpan.org/search?module=URI::Escape
#
use URI::Escape; #For uri_escape that
escapes characters before posting
use LWP::UserAgent; #Perl WWW library
use Digest::SHA1 qw(sha1 sha1_hex sha1_base64); #Required for SHA-1
authentication
use Digest::MD5 qw(md5 md5_hex md5_base64); #Required for MD5
authentication
############################
# check_valid_command
#
# Takes text string as input
# Returns true or false
############################
sub check_valid_command {
my $command = shift;
my @ARUBA_VALID_COMMANDS = (
"user_add", "user_authenticate",
"user_delete", "user_blacklist",
"user_query"
); #Ref "show aaa xml-api statistics"
foreach (@ARUBA_VALID_COMMANDS) {
return 1 if uc($command) eq uc($_);
}
#else
return 0;
}
############################
# check_valid_user_attribute
#
# Takes text string as input
# Returns true or false
############################
sub check_valid_user_attribute {
my $attr = shift;
my @ARUBA_VALID_USER_ATTRIBUTES =
( "ipaddr", "macaddr", "name", "role", "session_timeout" );
foreach (@ARUBA_VALID_USER_ATTRIBUTES) {
return 1 if uc($attr) eq uc($_);
}
#else
return 0;
}
############################
```

```
# check_valid_auth_method
#
# Takes text string as input
# Returns true or false
############################
sub check_auth_method {
#Note to add auth methods, you also need to add code to support the
generation of the <key> tag
#In aruba_generate_xml below, otherwise aruba_generate_xml will die
my $auth_method = shift;
my @ARUBA_VALID_AUTH_METHODS =
( "", "cleartext", "MD5", "SHA-1" ); #"" is no authentication
foreach (@ARUBA_VALID_AUTH_METHODS) {
return 1 if uc($auth_method) eq uc($_);
}
#else
return 0;
}
######################
# aruba_generate_xml
######################
# Takes a comand, a reference to a user array as input. Also passing
# an authentication method and password as extra input is optional.
# If present, these items are used to generate a authentication and
# key tags to send to the switch.
#
# Returns the xml text if there is no error. On error, returns
# "ERROR:" then the error message.
#
# Argument checking:
# Checks for a legal command and legal auth_method. Returns an error
# if they are not valid. All valid user attributes are passed into
# the XML while invalid attributes are discarded with a warning.
#
# Example usage:
#----8<----8<----8<----8<----
# my %user=(
# ipaddr => "10.215.1.6",
# name => "Mr Blibble",
# role => "logon"
# );
# my %auth=(
# method => "SHA-1",
# key => "supersecret",
# calculate => 1
# );
#
# #Switch/XML-client Authentication:
# my $xml_text = aruba_generate_xml ("user_add",\%user,\%auth);
#
# #No switch/XML authentication:
# my $xml_text = aruba_generate_xml ("user_add",\%user);
#----8<----8<----8<----8<----
######################
sub aruba_generate_xml {
my $ARUBA_VERSION = "1.0"; #This is included as is in the
<version>...</version> tags
```

```
my $num_args = scalar(@_);
return "ERROR: aruba_generate_xml needs at least 2 arguments: the
command and the user data"
if $num_args < 2;
my $command = shift;
my $user_ref = shift;
my $auth_method = "";
my $secret_key = "";
my $calc = 1;
#Get the extra arguments if present
if ( $num_args == 3 ) {
my $auth_ref = shift;
$auth_method = ${%$auth_ref}{'method'} if defined
${%$auth_ref}{'method'};
$secret_key = ${%$auth_ref}{'key'} if defined
${%$auth_ref}{'key'};
$calc = ${%$auth_ref}{'calc'} if defined
${%$auth_ref}{'calc'};
}
else {
if ( $num_args > 4 ) {
$auth_method = shift;
$secret_key = shift;
warn "Ignoring extra arguments to aruba_generate_xml";
}
}
my $hash = "";
#Check input:
return "ERROR: not a valid command: $command"
unless check_valid_command($command);
return "ERROR: not a valid authentication method: $auth_method"
unless check_auth_method($auth_method);
#Add the XML header:
my $xml = '<?xml version="1.0" encoding="ISO-8859-1" ?>' . "\n";
#Add the Aruba Command:
$xml .= "<aruba command=" . $command . ">\n";
#Add each valid user attribute in the user hash
foreach ( keys %$user_ref ) {
if ( check_valid_user_attribute($_) ) {
$xml .= "<$_>${$user_ref}{$_}</$_>\n";
}
else {
warn "Invalid user attribute $_ ignored.\n";
}
}
#Add the version
$xml .= "<version>$ARUBA_VERSION</version>\n";
#Add the Authentication method if present
if ( $auth_method ne "" ) {
$xml .= "<authentication>$auth_method</authentication>\n";
#If we're using cleartest, then just add it as is
if ( $auth_method eq "cleartext" ) {
$xml .= "<key>${secret_key}</key>\n";
}
else {
#If we need to calculate the hash, then use the right
method:
```

```
if ($calc) {
if ( $auth_method eq "MD5" ) {
$hash = md5_hex( ${secret_key} );
}
else {
if ( $auth_method eq "SHA-1" ) {
$hash = sha1_hex( ${secret_key} );
}
else { die "Unknown authentication method
${auth_method}.\n"; }
}
}
else { # if not $calc
#We're not calculating the has ourselves, so just use
it as it came in
$hash = $secret_key;
};
$xml .= "<key>$hash</key>\n";
}
}
#End with closing the aruba tag.
$xml .= "</aruba>\n";
return $xml;
}
#################
# aruba_post_xml
#
# Takes the switch address and the xml text as input
# Posts this information to the switch
# WARNING: This will die if called directly with an ERROR from
# aruba_generate_xml()
# Returns the response from the switch, or an error if occured.
#
#################
sub aruba_post_xml {
my $switch_addr = shift;
my $xml_text = shift;
#Check for errors in case the user doesn't.
die $xml_text if $xml_text =~ /^ERROR/;
#Make a new user agent
my $ua = LWP::UserAgent->new( agent => 'xml client' );
#And generate a POST request to the switch
my $req = new HTTP::Request POST =>
"https://${switch_addr}/auth/command.xml";
$req->content_type('application/xml');
#Note current versions require the xml_text to be prefaced with
"xml=" and escaped:
$req->content( "xml=" . uri_escape($xml_text) );
#Acctually perform the request and wait for the response:
my $resp = $ua->request($req);
#Return any error or the content otherwise.
return $resp->error_as_HTML unless $resp->is_success;
return $resp->content();
}
1;
```

Sample code to use the above procedures:

```
#!/usr/bin/perl -w
use strict;
my $SWITCH_IP = "10.215.1.206";
#Include the aruba routines:
require("./aruba_xml.pl");
############################
# Aruba Switch setting:
############################
#
# With xml-api client authentication:
#
# conf t
# xml-api client a.b.c.d key <secret>
#
# Without xml-api client authentication:
#
# conf t
# xml-api client a.b.c.d
#
# Where a.b.c.d is the IP address of the xml-api client.
#
# Also ensure that you have the Advanced AAA feature key enabled:
#
# (Aruba)# show keys
#
# Licensed Features
# -----------------
# Feature Status
# ------- ------
# ...
# Advanced AAA Services ENABLED
############################
# How to use aruba_xml.pl:
############################
#
# Step 1. (Mandatory)
# Generate a user hash:
#
# Advanced users, note that these routines generate xml tags and data
# directly from this hash by checking the hashes key against valid
# tags and including that key if valid
# Current valid keys are (ref: @ARUBA_VALID_USER_ATTRIBUTES )
# "ipaddr" : User's IP Address
# "macaddr" : User's MAC Address
# "name" : User's Name
# "role" : Users's Role
# "session_timeout" : User Timeout
my %user = (
ipaddr => "10.9.8.7",
name => "Mr Blibble",
role => "logon"
);
############################
#
# Step 2. (Optional)
# Specify Authentication
#
```

```
# This authentication is for the xml-client to switch authentication
# It is highly recommended to enable.
#
# If the switch has a key set, then without specifying the
# authentication here, you'll just get errors.
#
# Hash keys:
# method: method can be any of "cleartext", "MD5" or "SHA-1".
# specifies what type of hash
# calc: specifies whether or not to calculate the hash
# 0: Do Not calculate the hash. This means the key is
# already a hash and will be placed in the xml verbatim
# 1: Do calculate the hash. The default if this attribute
# is not present.
# key: The secret key or hash (see calc)
# Example 1:
# A pre-calculated SHA-1 hash. The recommended usage.
# To calculate the hash, try "sha1sum" from a unix box
my %fast_auth = (
method => "SHA-1",
key => "a761ce3a45d97e41840a788495e85a70d1bb3815", #This is the
sha-1 hash of "supersecret"
calc => 0 #This is a hash, so don't calculate it.
);
# Example 2:
# Without pre-calculating the hash. Use only for testing
my %slow_auth = (
method => "SHA-1",
key => "supersecret"
);
# Example 3:
# Cleartext authentication
my %clear_auth = (
method => "cleartext",
key => "supersecret"
);
################################
#
# Step 3. (Mandatory)
# Call aruba_generate_xml to generate your xml
#
# Give it the command, one of:
# "user_add" : Add a user to the switch
# "user_authenticate" : Force the switch to authenticate a user
# "user_delete" : Delete a user from the switch
# "user_blacklist" : Blacklist a user (force a de-auth and do
# : not allow them access)
# "user_query" : Query the switch about the user
#
# Also give it the reference to the user hash to do the above command
# And finally the reference to the authentication hash if present
#
#
# Step 4. (Mandatory)
# Call aruba_post_xml to post it to the switch.
# Give it the ip or domain name to send to the request to the switch
# And finally the xml text as well.
```

```
############################
# Example 1. Recommended Method
#--------------------------
# For example, "user_add" command with authentication:
#
# The recommended and safe method:
# User add example with error checking and pre-calculated hash
# Note how the user and auth hash are passed as a reference
if (1) {
# Step 3: Generate some XML text to add the above user with the
required auth method
my $xml_text = aruba_generate_xml( "user_add", \%user, \%fast_auth
);
# Check for errors:
die $xml_text if $xml_text =~ /^ERROR/; #Errors are returned by
starting with ERROR
#Uncomment the next line to dump the xml text to the screen
# print "XML text to post:\n".$xml_text;
# Step 4: Post the XML to the switch
print "Sending Example 1 request:\n";
my $response = aruba_post_xml( $SWITCH_IP, $xml_text );
# Check for errors:
#We have a problem if the switch didn't return
"<status>Ok</status>" in it's response
die "Error detected in response from switch:\n$response"
unless $response =~ m|<status>Ok</status>|i;
print "Switch Status OK!\n";
print $response;
}
############################
# Example 2.
#--------------------------
# Simple blacklist no auth, no error checking
# This is not recomended, because you should do your own
# error checking, but it is very simple and if you're OK
# with the code dying in the case of user/auth errors, you
# should be OK.
#
# Steps 3 and 4 are combined into one line
#
# Be prepared to die if the command or auth parameters are invalid.
if(0) {
print "Sending Example 2 request:\n";
print aruba_post_xml( $SWITCH_IP, aruba_generate_xml(
"user_blacklist", \%user ) );
};
############################
# Example 3.
#--------------------------
# Simple user query, with SHA-1 used to authenticate
# this xml-client, but no pre-calculating of the hash
#
# Be prepared to die if the command or auth parameters are invalid.
if (1) {
print "Sending Example 3 request:\n";
print aruba_post_xml( $SWITCH_IP, aruba_generate_xml( "user_query",
\%user, \%slow_auth ) );
```

```
}
```

5 Examples

Configuration
==========
aaa xml-api client 10.4.0.61
key itsabug (max 48 char)
!
show commands
==========
show aaa xml-api clients
show aaa xml-api stat

Use ecp to test XML interface:

1. Add user
   # ecp -i 10.4.35.208 -t 30 -a md5 10.4.0.35 add itsabug
2. Delete user (only delete by IP works)
   # ecp -i 10.4.35.208 10.4.0.35 delete itsabug
3. Query user
   # ecp -i 192.168.5.105 -a md5 192.168.5.31 query itsabug
4. Blacklist
   # ecp -i 10.4.35.212 -a md5 10.4.0.35 blacklist itsabug
5. Authenticate
   # ecp -i 10.4.35.208 -n bchoy -p 1 -a md5 10.4.0.35 authenticate itsabug

NOTE:

1. Mixed query is not supported, e.g. IP + role
2. IP address is minimal required field for all XML commands