



CHAPTER 15

Using the Scripting Interface (CSAAPI)

Overview

The Management Center for Cisco Security Agents provides the ability for administrators to write scripts to perform a subset of configuration actions on the MC.

Sample scripts are provided with the MC and it is recommended that you use these samples for writing your own scripts.

This section contains the following topics.

- [Overview, page 15-1](#)
- [CSAAPI/Scripting Overview, page 15-2](#)
- [API Function Summary, page 15-3](#)
- [Scripting Interface Fundamentals, page 15-4](#)
- [Before You Begin, page 15-4](#)
- [WSDL/SOAP, page 15-5](#)
- [Choosing a Scripting Language, page 15-6](#)
- [Sample Scripts and README Files, page 15-7](#)
- [Encryption and Authentication, page 15-8](#)

- [Object Expressions, page 15-9](#)
- [Object Types, page 15-10](#)
- [Supported Names per Object Type, page 15-10](#)
- [Object Expression Values, page 15-12](#)
- [Object Type Names, page 15-16](#)
- [Wildcarding, page 15-17](#)
- [Using the Escape Character, page 15-17](#)
- [Using the Limit Name to Prevent Unwanted Actions, page 15-19](#)
- [API Function Descriptions, page 15-21](#)
- [Getting the Status of Functions and Waiting for Functions to Complete, page 15-21](#)
- [Testing Object Expressions, page 15-23](#)
- [Modifying the State of the Overall System, page 15-24](#)
- [Host Group Assignment, page 15-27](#)
- [Manipulating Hosts, page 15-29](#)
- [Getting Host Information, page 15-30](#)
- [Getting Overall System Information, page 15-32](#)
- [Getting Event Information, page 15-33](#)
- [Getting Reports, page 15-33](#)

CSAAPI/Scripting Overview

The Management Center for Cisco Security Agents provides administrators with the ability to use scripts for performing some very specific functions locally on the MC or from a remote system. This scripting interface is called CSAAPI and it allows you to write your scripts in any language and run your scripts on any operating system.

API Function Summary

The CSA MC scripting interface lets you execute a variety of functions, which include manipulating hosts and retrieving host information. The following is a list of actions that are possible:



Note

Before attempting to run the functions listed here, read through the [Scripting Interface Fundamentals, page 15-4](#) and [API Function Descriptions, page 15-21](#) sections so that you understand the general concepts and individual API functions listed here.

Non-blocking Functions

- Integer GenerateRules ()
- Integer GenerateReport (String reportName, String emailAddress)
- Integer Import (String filename)
- Integer Export (String filename, String desc, String emailAddress, StringArray objects)
- Integer RunHostTask (String hostTaskName)
- Integer BackupConfig ()

Blocking Functions

“Get” Functions

- StringArray GetHostInfo (String hostExpression)
- StringArray GetLastIpAddr (String hostExpression)
- StringArray GetDiagnosticInfo (String hostExpression)
- StringArray GetMostActive (String objType, Integer numOfObjects, String timespan)
- Integer GetLastDayEventCount ()
- StringArray GetLatestEvents (String objType, String objExpression, Integer numOfEvents)
- StringArray GetLatestAudit (Integer numOfEvents)
- String GetConfigVar (String name)

- Integer GetObjectCount (String objType, String objExpression)
- StringArray GetNames (String objType, String objExpression)
- StringArray GetReportNames ()
- StringArray GetHostTaskNames ()

“Set/Update” Functions

- Boolean AddHostsToGroups (String hostExp, String groupExp)
- Boolean RemoveHostsFromGroups (String hostExp, String groupExp)
- Boolean MoveHosts (String hostExp, String fromGroupExp, String toGroupExp)
- Boolean DeleteHosts (String hostExp)
- Boolean SendHint (String hostExp)
- Boolean ResetAgent (String hostExp, Integer resetMask)
- Boolean SetConfigVar (String name, String value)

“Housekeeping” Functions

- Integer GetStatus (Integer processID)
- String GetStatusMessage (Integer processID)
- Integer WaitForProcess (Integer processID, Integer timeout)

Scripting Interface Fundamentals

The following sections provide information you need to use the scripting interface.

Before You Begin

Before you attempt to write scripts that use the CSA MC scripting interface, it is critical that you take some time to familiarize yourself with a number of important concepts described in the next several sections, including:

- WSDL/SOAP
- Choosing a scripting language
- Using the Sample Scripts and README files
- Encryption and Authentication
- Object Expressions
- Wildcarding
- Using the Escape Character
- Using the LIMIT token to prevent unwanted actions
- Blocking vs. Non-blocking Functions
- Getting the status of functions and waiting for functions to complete
- Getting to know each scripting interface function

Bear in mind that the scripting interface is a powerful tool that allows you to execute actions that can have serious consequences. For example, the scripting interface allows you to perform tasks such as deleting hosts, resetting agents, generating rules and modifying host/group assignments. As such, it makes sense to exercise caution while using the scripting interface and take the time to understand its intricacies.

WSDL/SOAP

The CSA MC scripting interface infrastructure relies upon WSDL (Web Services Description Language) and SOAP (Simple Object Access Protocol) for its function. WSDL and SOAP allow you to use widely-available, non-proprietary libraries to write scripts for most operating systems in a variety of different languages. In short, SOAP and WSDL allow you to write distributed applications without having to understand the complexities of the communication between your scripts and the CSA MC server.

That said, before writing your scripts, it is recommended that you take a few moments to familiarize yourself with the basics of WSDL and SOAP. The sample scripts will be easier to follow and debugging may be easier with just a few moments devoted to WSDL and SOAP.

WSDL is a specification for defining web services/APIs with a common XML grammar. WSDL is powerful because it abstracts out details relating to specific operating systems and computer languages. CSA MC contains a WSDL file (csaapi<csa mc version number>.wsdl, for example csaapi52.wsdl) that describes which functions are available, what parameters they take, and what values they return. This WSDL file can be used by a variety of freely-available utilities to generate script “stubs” that can be quickly modified into working scripts. (Or you can just start with the sample scripts provided with CSA MC – more on this later.)

Good information on WSDL can be found here:

<http://www.w3.org/TR/wsdl>

<http://www.oreilly.com/catalog/webservess/chapter/ch06.html>

<http://en.wikipedia.org/wiki/WSDL>

SOAP is an XML-based protocol for object exchange (usually over HTTP). While WSDL is used to describe web services, SOAP is a protocol for encapsulating messages (requests and responses, usually) in an “envelope” that is communication-protocol independent. SOAP has been widely adopted and there are a variety of good SOAP implementations (for example, Perl’s SOAP::Lite) that make writing web-based scripts easier than you would think.

Good information on SOAP can be found here:

<http://www.w3.org/TR/soap12-part1/>

<http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>

http://en.wikipedia.org/wiki/Simple_Object_Access_Protocol

Choosing a Scripting Language

As is stated above, WSDL and SOAP allow you to write scripts in any language that supports WSDL and SOAP, for any platform on which the appropriate libraries are available.

That being said, there are sample scripts provided for you in the following languages: Perl, Java and C++. It is *highly* recommended that you start with the sample scripts and understand their workings before attempting to move onto other languages (for example, C# or PHP). It’s recommended that you start with the sample scripts because the scripts contain examples of many different

techniques, from asking for passwords, to using the LIMIT clause, to checking the status of functions and getting status messages, to using wildcards and escape characters, to using object expressions (all of which are explained here).

Furthermore, if you do not have a language preference, it's recommended that you choose Perl for your first workings with the scripting interface. The Perl infrastructure, including its encryption libraries, is easy to install and use. Perl, as an interpreted language, lends itself to fast prototyping and has nice features for running in debug mode that make debugging easier than in the other languages.

Finally, please bear in mind that Cisco TAC can assist you if there is a problem with the scripting interface infrastructure, but *Cisco TAC will not help you debug your own scripts.*

Sample Scripts and README Files

CSA MC installs with sample scripts which you should use to model your own scripts. These sample scripts appear in a separate folder for each scripting language. These folders also contain other files that are meant to help you write your scripts, including a detailed readme file. The sample scripts can be found in the “samples\csaapi” directory of your CSA MC installation directory, for example:

```
C:\Program Files\Cisco Systems\CSAMC\CSAMC52\samples\csaapi
```

This directory contains a README.txt file that contains general information on the Scripting Interface. Furthermore, this directory contains “Perl”, “Java” and “CPP” directories that contain sample scripts for each language. Note that each language directory contains its own README.txt file that contains important information specific to each language.



Caution

It is critical that you read these README.txt files before attempting to create or run a script. The readme files contain specific information for setting up your MC server environment and your client environment. This information is not provided anywhere else and you must follow the instructions provided in the readme for your scripts to work as expected.

Encryption and Authentication

In order to ensure that the CSA MC scripting interface is not used in an unauthorized manner, the scripting interface mandates the use of encryption and authentication.

Encryption is provided by the use of the *https* protocol. In the sample scripts, you will see an URL embedded with “https” as the protocol. Do not try to change this protocol as the scripts will not work.

Authentication is provided outside the CSA MC user-interface login system. The scripting interface does not have knowledge of existing administrator accounts on the MC. Before you can use the scripting interface, you must create a new userid and password using the instructions provided in this section in order to use a script to configure the MC. The userid and password you create to access the MC with your script will have no bearing on the CSA MC user interface access and will only apply to the scripting interface.

Before you use the scripting interface, you *must* run through the following steps to provide the scripting interface infrastructure with userid and password information:

First, go to the apache directory by typing:

```
cd <CSAMC_base_directory>\apache2
```

For example:

```
cd Program Files\Cisco Systems\CSAMC\apache2
```

Then run the `htpasswd` utility with the following syntax:

```
bin\htpasswd.exe -c conf\csaapiUsers<csa mc version number>.pwd <username>
```

You will be prompted to enter that user's password two times.

To add additional users, type the following:

```
bin\htpasswd conf\csaapiUsers<csa mc version number>.pwd <next user name>
```

Note the file **MUST** be named `csaapiUsers.pwd`.

Once this is done, your scripts must provide one of these userid/password combinations. See the README files in the `csaapi` directory and the sample scripts for details.

Object Expressions

Many functions in the CSA MC scripting interface require you to specify which objects to act on. For example, the “DeleteHosts” function requires you to specify which hosts to delete.

**Note**

An “object”, for the purposes of the scripting interface, is an entry in the CSA MC database corresponding to an entity like a host, group, policy, rule module, rule, etc.

To provide maximum power to describe groups of objects, the CSA MC scripting interface uses an “object expression” syntax, which is simply a set of name/value pairs with the format:

```
NAME='VALUE' NAME='VALUE' NAME='VALUE'
```

For example, consider the following scripting interface function:

```
StringArray DeleteHosts (String hostExpression)
```

To delete all Windows hosts in Learn Mode, for example, you would call the function with the following argument:

```
DeleteHosts ("OS='Windows' LEARN_MODE='true'");
```

The object-expression syntax is basically an “SQL Lite” system that is both flexible and independent of the CSA MC database schema. Multiple Name/Value pairs are logically ANDed together. In other words, the example above would apply only to hosts that are both Windows AND in Learn Mode.

If an invalid expression is passed to any function, the function will return an error code, and the caller can use the `GetStatus()` and `GetStatusMessage()` functions to get more information.

Now that the syntax has been explained, let’s take a look at what names are valid, and what values are valid for each name.

Object Types

There are some general issues to note when using a scripting interface to access CSA MC. You cannot create new configuration items using the scripting interface. You can only manipulate existing items and you can also generate rules. This next section provides information, universal to all scripting languages, on the CSA MC functions (object types) that are available for configuration via scripting. Each object is specified in terms of name/value pairings. The following is a list of supported object types:

```
HOST
GROUP
POLICY
RULE_MODULE
RULE
APP_CLASS
VAR
```

Supported Names per Object Type

The following describes which names are valid for which object types:

Object Type: Hosts

Names:

```
HOST_NAME
IP_ADDR
GROUP_NAME
POLICY_NAME
RULE_MODULE_NAME
OS_TYPE
OS_ARCHITECTURE
OS_DESC
ACTIVE
TEST_MODE
LEARN_MODE
```

Object Type: Groups

Names:

```
GROUP_NAME
POLICY_NAME
```

HOST_NAME
VERSION
TEST_MODE
LEARN_MODE
OS_TYPE
OS_ARCHITECTURE

Object Type: Policies

Names:

POLICY_NAME
GROUP_NAME
RULE_MODULE_NAME
VERSION

Object Type: Rule Modules

Names:

RULE_MODULE_NAME
POLICY_NAME
VERSION
TEST_MODE
LEARN_MODE
OS_TYPE
TARGET_OS

Object Type: Rules

Names:

RULE_ID
RULE_NAME
RULE_TYPE
ACTION
RULE_MODULE_NAME

Object Type: Application Classes

Names:

APP_CLASS_NAME
VERSION
OS_TYPE
TARGET_OS

Object Type: Variables

Names:

VAR_NAME
 VAR_TYPE
 VERSION
 OS_TYPE
 TARGET_OS

Object Expression Values

Valid values for each of the names listed in the previous section are listed here. The use of wildcard characters is supported in all value entries unless otherwise specified:

HOST_NAME

A string expression of a host name would appear as follows. Value examples:

```
bugtrack.amer.cisco.com
bugtrack*
bugtrack.????.cisco.com
```

Note that host name values are NOT case sensitive.

IP_ADDR

A string expression of an IP address would appear as follows. Value examples:

```
1.1.1.1
1.1.1.*
15?.*.*.*
```

GROUP_NAME

A string expression of a group name would appear as follows. Value examples:

```
Desktops - All types
Servers - * deployed
```

Note that group name values are NOT case sensitive.

POLICY_NAME

A string expression of a policy name would appear as follows. Value examples:

```
Email client - Linux
Email client *
```

RULE_MODULE_NAME

A string expression of a rule module name would appear as follows. Value examples:

```
Email Client Module - Low Security
Email Client Module - ? Security
```

RULE_ID

A string expression of a rule ID would appear as follows. Value examples:

```
"51"
```

RULE_NAME

A string expression of a rule name would appear as follows. (Note that rules, technically, do not have names. **RULE_NAME** gives you the ability to query against the Description field that is displayed in the MC UI for an individual rule.) Value examples:

```
Email worm
```

RULE_TYPE

This field lets you identify a rule by its type. Since there is a specific name for each rule type, a rule type value you enter must match one of the following strings:

```
Agent service control
Agent UI control
Application control
Buffer overflow
Clipboard access control
COM component access control
Connection rate limit
Data access control
File access control
Global event log
Global IP address quarantine
Global network scan
Global virus scan
Kernel protection
Network access control
Network interface control
Network shield
NT Event log
Registry access control
```

```

Resource access control
Rootkit / kernel protection
Service restart
Sniffer and protocol detection
Syslog control
System API control

```

Rather than trying to match the string exactly, you can also use wildcards to match types. For example, the following would be less likely to be foiled by a typo:

```
RULE_TYPE='Rootkit*'
```

than this entry:

```
RULE_TYPE=' Rootkit / kernel protection'
```

ACTION

The following string values are allowed for this name. Since there are a specific number of action types, the value you enter must match one of the following strings. Any other value results in an error:

```

PRIORITY_TERMINATE
PRIORITY_DENY
PRIORITY_ALLOW
QUERY
TERMINATE
DENY
MONITOR
SET

```

Note that wildcards are not supported for ACTION types.

VERSION

A string expression of a CSA MC release version would appear as follows. Value examples:

```

5.2 r578
5.2*
5.2.? r6??

```

OS_TYPE

This allows you to specify either 'WINDOWS' or 'UNIX'. Generally, this field correlates to the "ostype" field in the various database tables.

OS_ARCHITECTURE

This field provides a way to specify more granularity than with the `OS_TYPE` field. This field can be applied to objects of type `HOST` and `GROUP`. The valid values are 'WINDOWS', 'SOLARIS', and 'LINUX'. In general, the field corresponds to the "architecture" field in the database.

TARGET_OS

This field provides a way to specify more granularity than with the `OS_TYPE` field. This field can be applied to objects of type `RULE_MODULE`, `APP_CLASS` and `VAR`. The valid values are 'ALL', 'XP', 'SOLARIS', and 'LINUX'. In general, the `TARGET_OS` field corresponds to the "osflavor" field in the database

OS_DESC

This field lets you query directly on the host's "os" database field. This contains descriptions such as "Windows 2003" that could allow you to differentiate between Windows 2000 and Windows 2003.

ACTIVE

For this name, you can enter a boolean value ('TRUE' or 'FALSE') corresponding to whether the machine is active. 'YES' and 'NO' are also acceptable values.

TEST_MODE

For this name, you can enter a boolean value ('TRUE' or 'FALSE') corresponding to whether the machine is active. 'ON' and 'OFF' are also acceptable values.

LEARN_MODE

For this name, you can enter a boolean value ('TRUE' or 'FALSE') corresponding to whether the machine is active. 'ON' and 'OFF' are also acceptable values.

APP_CLASS_NAME

A string expression of an application class name would appear as follows. Value examples:

```
FTP applications
Software installer invoked applications
```

VAR_NAME

A string expression of a variable name would appear as follows. Value examples:

```
Apache executables
Ephemeral port ranges
```

VAR_TYPE

The following string values are allowed for this name. Since there are a specific number of variable types, the value you enter must match one of the following strings. Accepted values are:

```
COM
DATA
FILE_UNIX
FILE_WINDOWS
NETWORK_ADDRESS
NETWORK_SERVICE
QUERY
REGISTRY
```

Note that wildcards are not supported for VAR_TYPE.

Object Type Names

There are several functions in the scripting interface that are generic and that can be run for objects of any type (`GetNames()`, `GetMostActive()`, etc.). The scripting interface uses a string variable typing system to indicate to these generic functions the type of object being acted upon. The following are accepted values:

```
HOST
GROUP
POLICY
RULE_MODULE
RULE
APP_CLASS
VAR
```

The listed object type values are special because they can be passed separately into functions requiring an “object type” string, but they can also be combined with the name “OBJ_TYPE” in a name/value pair for any object type. For example, to get a list of names of Application classes associated with Linux, you could create the following name/value expression:

```
“OBJ_TYPE=‘APP_CLASS’ OS=‘Linux’”
```

The OBJ_TYPE designation can be omitted for any function in which the object type is implied by the function itself (for example, `GetHostInfo`) or any function that takes as a parameter an object-type string that is separate from the object expression (for example, `GetObjectCount`).

Wildcarding

The scripting interface provides wildcard support similar to what is available through SQL, although with slightly different syntax. The following wildcard characters are supported:

- * Matches any string of zero or more characters
- ? Matches any one character

Using the Escape Character

Although this will be rare, there may be times when you want to specify a wildcard character (or the single-quote character) literally. In other words, when you want to express a string that has the wildcard character in it, for example:

```
Bob's Group
```

In order to support expressions like this, there are four characters that must be “escaped” with the “\” escape char. These characters are:

- *
- ?
- ' (single quote)
- \ (backslash)

Therefore, if you wanted to specify a Group Name of:

```
Bob's Group
```

You would need to place a backslash before the single-quote, for example:

```
GROUP_NAME='Bob\'s Group'
```

Literal backslash characters also require a backslash. For example, to specify the Group Name `C:\foo`, you would need to enter:

```
GROUP_NAME='C:\\foo'
```

As a final example, if you wanted to specify a string ending in a question mark, the user would need to enter:

```
GROUP_NAME='*\?'
```

In the above example, the asterisk is a wildcard, but the question mark is used literally.

It should be noted that CSA MC provides restrictions on which characters can be present in names of groups, hosts, etc. As such, you will rarely have to use a wildcard character literally. But the escape-character functionality exists in the event that CSA MC naming restrictions are lifted in the future.



Note

Note to Perl Script Writers: Perl provides an extra hurdle when using the backslash character. In order to embed the backslash character in a string in Perl, you must unfortunately specify two backslashes. So, in Perl, to specify the group name `Bob's Group`, you must actually provide text like this:

```
my $objExp = "GROUP_NAME='Bob\\'s Group'"
```

In a more painful example, to specify `C:\foo`, you will actually need to specify four (4) backslashes, as in:

```
my $objExp = "GROUP_NAME='C:\\\\foo'"
```

This is an unfortunate consequence of having to escape the backslash character twice – once for Perl and once for the scripting interface. For more details, see the Scripting Interface README files.

Using the Limit Name to Prevent Unwanted Actions

There is one name token that can be applied to objects of any type. That token is called `LIMIT`. The `LIMIT` construct allows the caller to specify that a function must terminate immediately if the number of objects returned by the function's object expression exceeds the specified limit.

For example, if you want to delete all hosts that begin with the string "server", but your assumption is that no more than two hosts meet this description, you could run the following:

```
DeleteHosts ("HOST_NAME='bugtrack*' LIMIT='2'")
```

This call would delete no hosts if more than two hosts started with the string "server".

This `LIMIT` construct in effect allows you to call the `GetObjectCount` method within a called function, saving an extra call. It should be noted that the `LIMIT` token has no effect if placed within the `GetObjectCount` function.

In the event of a failure of a function that has an object expression that contains a `LIMIT` statement, call the `GetStatus()` function (described below). If the error code is "200", then this means that the number of objects described by the object expression exceeds the `LIMIT` value.

Blocking vs. Non-blocking

Generally, Scripting Interface functions can be split into two groups: Those that "block" and those that don't. The following section describes the difference and how this difference relates to return values and getting status information.

Blocking Functions

When a blocking function is executed, the server waits until the function is complete before returning a value. All but a handful of functions in the API are blocking.

Generally, blocking functions that implement a “set/update” operation (for example, `AddHostsToGroups`) return a boolean value (`false` on failure and `true` otherwise). Blocking functions that implement a “get” operation (for example, `GetLastIpAdrs`) return the data type of whatever data is being retrieved (string, string array, integer, etc).

Detailed status/error information of a blocking function can be retrieved by calling one of the status functions (`GetStatus` and `GetStatusMessage`) with the Process ID of 0. See the section *Getting the Status of Functions and Waiting for Functions to Complete* for more information.

Non-blocking Functions

Non-blocking functions return control back to the client script before the function is complete. Generally, functions that can take a long time to complete are the non-blocking functions. Non-blocking functions return control back to the client script immediately to ensure that the connection between the client and server does not time out. The current list of non-blocking functions is as follows:

```
GenerateRules()
GenerateReport()
Import()
Export()
RunHostTask()
BackupConfig()
```



Note

All other functions in the scripting interface block.

When a non-blocking function runs, the `scriptinginterface` server basically tells the client script “I got your request, and I am working on it. Please check back with me later to learn how it went.” All non-blocking functions return an integer Process ID that can be used to get the status of the function later.

Client scripts can use the `WaitForProcess` function to wait until a non-blocking function is complete and can use the status functions (`GetStatus` and `GetStatusMessage`) to check on a non-blocking function’s status. See the section *Getting the Status of Functions and Waiting for Functions to Complete* for more information.

API Function Descriptions

The following sections provide more information for each individual API function.

Getting the Status of Functions and Waiting for Functions to Complete

The CSAMC Scripting Interface provides a comprehensive infrastructure for retrieving the status of functions that have recently been executed. This section describes the infrastructure's main elements, which include numeric *status codes*, *process IDs*, and *functions* that return status information and wait for non-blocking functions.

Status Codes

The Scripting Interface uses numeric status codes to communicate the status of a function that has been executed by a client. In general, a status code of zero reflects success, a status code greater than zero reflects failure, and a status code of less than zero indicates that nothing has happened.

The complete list of status codes is given below:

Success		Failure		No Action	
ID	Description	ID	Description	ID	Description
0	SUCCESS	1	GENERIC ERROR	-1	PENDING
		100	UNKNOWN PROCESS	-100	NO ACTION
		200	ILLEGAL OBJECT EXPR.	-200	NO OBJECT MATCH
		300	DB OPEN FAILURE	-201	NO OBJECT MATCH
		301	DB READ FAILURE	-300	LIMIT EXCEEDED

Most of these status codes are self explanatory, but a few require elaboration, below.

If you pass to a function an improperly formed object expression, then this situation is an error condition and the status code for that situation will be 200.

If, on the other hand, you pass to a function an object expression that happens to match no objects (for example, if your object expression is `"HOST_NAME='A*'"` but you have no hosts that begin with the letter 'A'), then you will receive the status code -200 (negative 200), which is not an error code – it's a no-action code. In general, *the scripting interface does not consider a request to do nothing to be an error.*

Similar to the situation above, if you specify a LIMIT clause in an object expression and the number of objects described by the object expression exceeds the limit specified in the LIMIT clause, then the return code will be -300 (negative 300), which, again, is not an error code but rather a no-action code.

Process IDs: Identifying the Execution of a Function

Each time you run a function, a Process ID is associated with the execution of that function. This Process ID can then be used to get the status code association with the function execution.

The Process ID for a non-blocking function (see the [Blocking vs. Non-blocking](#) section for details) is the integer returned by the function.

The Process ID for a blocking function is always zero. In other words, getting the status code for the process with a Process ID of zero is the same as getting the status code for the last blocking function that was run.

Getting a Status Code for an Execution of a Function

Once you have the Process ID for a function you have run, getting the status of that function is simply a matter of passing the Process ID to the following function:

```
- Integer GetStatus (Integer processID)
```

This API function asks the scripting interface server for the appropriate status code. The argument is the Process ID, and the returned value is the status code.

Getting a Detailed Status Message

To get text information about the specific nature of an error, call the following function:

- `String GetStatusMessage (Integer processID)`

This function returns the status message of the process with the passed-in ID. If the passed-in ID is zero, then `GetStatusMessage` returns the status message of the last API function that was called.

If the process with the passed-in ID is still running, then the returned value will be “PENDING”. If the process succeeded, then the returned value will be “SUCCESS”.

`GetStatusMessage` is often helpful when attempting to receive additional information about why a function failed.

Waiting for a Non-blocking Function to Complete

- `Integer WaitForProcess (Integer processID, Integer timeout)`

This function *blocks* until the specified process is complete or until the timeout (expressed in seconds) expires, whichever comes first. This function returns the Status Code of the specified function if the function finishes before the timeout or -1 otherwise. This function provides a handy way of blocking for function completion without having to write a loop.

Testing Object Expressions

Object expression strings are used to describe a set of objects and are passed to functions as arguments. But some functions can have devastating effects if run against the wrong set of objects. The following two menu functions provide “sanity-checks” that allow scripts to determine what objects are returned by a particular expression before running an operation on that expression.

```
Integer GetObjectCount (String objType, String  
objExpression)
```

`GetObjectCount` returns the number of objects referenced by the object expression. For example:

```
GetObjectCount (“HOST”, “OS=’WINDOWS’ ”)
```

would return the number of Windows hosts.

**Note**

The `LIMIT` token has no effect in this function and generally you should avoid including it in the object expression for this function.

```
StringArray GetNames(String objType, String
objExpression)
```

`GetNames` returns a string array containing the names of all the objects matching the object expression. For example:

```
GetNames ("HOST", "OS='LINUX' ACTIVE='FALSE'")
```

would return a list of hostnames of all inactive Linux boxes.

**Note**

For the object type “RULE”, this function returns the string form of the rule ID because rules do not technically have names.

Modifying the State of the Overall System

The following functions modify the state of the overall system. Each function kicks off a process that executes the intended action and then immediately returns a Process ID. These functions do NOT block until their work is done.

```
Integer GenerateRules ()
```

`GenerateRules()` simply generates rules. This function does not block. It returns a Process ID that can be used by `GetStatus` to determine whether the rule generation was successful.

```
Integer Import (String filename)
```

`Import()` imports objects from the specified file. A full pathname must be provided. `Import` does not block. It instead returns a Process ID. Note that wildcards are NOT supported in the filename.

**Note**

You cannot import from files on remote drives. Import files must reside locally on the CSA MC system.

```
Integer Export (String filename, String desc, String
emailAddr, StringArray objExpressions)
```

The `Export()` function exports objects to a file. The exported file is saved to the default CSAMC export directory and the file is accessible through the Exports page on the CSAMC GUI. If an optional email address is provided, then a copy of the export file is sent in an attachment to the email address.

The objects are specified by providing an array of object expressions. Each object expression is in the form of the name/value pairs described in the *Object Expressions* section.

Important: It is critical that each object expression provide an object type (for example, “OBJ_TYPE='POLICY'”). An error will result otherwise.

A sample array to export all Windows Application Classes would be:

```
sampleArray[] = { "OBJ_TYPE='APP_CLASS' OS_TYPE='W' " };
```

`Export` does not block; rather, it returns a Process ID.

Important Note: Not all possible object expressions are permissible in the `Export()` function. You cannot export objects of type `HOST` or `RULE`. Also, the only permissible name following an object of type `VAR` is `VAR_TYPE`. In other words, you cannot export variables by name, version or `OS_TYPE` or `OS_FLAVOR`. The only way variables can be exported is by `VAR_TYPE`.

Note: In order for this function to send email, the scripting interface must be informed of the name of the server to which to send emails. To do this, use the `SetConfigVar` function to set the “`csaapi.email_server`” parameter to the name of the email server. Note that this must be done only once.

Note: To allow the scripting interface to send email, you must first make a minor modification to the CSAMC rule that governs permissible web server actions. Please see the scripting interface sample script `README.txt` files for details.



Note

To email an export file to multiple addresses, separate the email addresses with a semicolon (;). For example: `joe@company.com;jane@company.com`

```
Integer RunHostTask (String hostTaskName)
```

`RunHostTask()` executes a Host Managing task. `RunHostTask` does not block; rather, it returns a Process ID. `RunHostTask` supports the CSA MC scripting interface wildcarding mechanism.

```
Integer BackupConfig()
```

`BackupConfig()` executes a backup task. `BackupConfig` does not block; rather, it returns a Process ID. Note that `BackupConfig` will fail unless the user has previously specified a backup type and a Directory name on the Maintenance>Backup Configuration screen in the CSA MC UI.

```
Boolean SetConfigVar(String name, String value)
```

`SetConfigVar()` performs the same function as the “`set_config.csapl`” script on the CSA MC machine. `SetConfigVar()` allows you to modify a number of system-wide, persistent CSA MC configuration variables. In general, documentation of the valid config variable names and their associated values is located elsewhere in the CSA MC documentation, but the following two config variables are documented here because of their importance for all scripting interface functions that send emails:

```
csaapi.email_server
```

```
csaapi.orig_email_addr
```

The first config variable is used to specify to which address e-mails are sent by the scripting interface. The second config variable is used to specify the originator’s email address for emails sent by the scripting interface.

For example, to tell the scripting interface to send all emails (say, from `GenerateReport`) to the server `myserver.cisco.com`, make the following call:

```
SetConfigVar ("csaapi.email_server",
"myserver.cisco.com")
```

Again, please note that the email server **MUST** be set before running any scripting interface function that generates an email.

Note: Wildcards are NOT supported in any of these string parameters.

Note: All configuration variables set with this function are persistent in the CSA MC database. In other words, you only need to set them once.

```
String GetConfigVar (String name)
```

`GetConfigVar` returns the value of any configuration variable stored in the CSA MC configuration database. It returns the same values as the “`set_config.csapl`” script on the CSA MC machine.

See the `SetConfigVar` documentation, above, for more information about these CSA MC configuration variables.

Note: An empty string can be returned as the result of three distinct situations: Either the value is indeed the empty string in the database, or the value is not set at all in the database, or an error occurred in the Scripting Interface. To distinguish between these three cases, call `GetStatus`. A zero return code (success) indicates that the value is indeed the empty string in the database. A negative return value (no action) indicates that there was no matching record in the database. A positive return value (failure) indicates that there was an error in the Scripting Interface, and that `GetStatusMessage` should be called to learn more about the error.

Host Group Assignment

The following functions change the Host/Group assignment. In all of the following functions, TRUE is returned on success and FALSE is returned on failure. Also, for all these functions, a failure with any part of the function will result in a rollback of any changes made. In other words, either all hosts matched by the `hostExpression` will be properly manipulated, or none will.

These functions block until the operation is complete. Note that these functions do not generate rules. That must be done separately.

For all of these functions, it is not considered an error/failure if the host or group expressions return no hosts/groups. In this case, the functions simply return TRUE (success) and do nothing. The reasoning for this is that scripts may simply want to manipulate certain hosts that meet certain criteria, but it may be common for no hosts to meet the object expression criteria at any given time.

If a script wants to detect the case in which no hosts were manipulated because no objects matched the object expressions, the script can call the `GetStatus` function.

```
Boolean AddHostsToGroups (String hostExp, String groupExp)
```

This function adds ALL the hosts matched by the host expression to ALL the groups matched by the group expression. This function does not remove the hosts from any groups. It just adds.

The following example adds all hosts in the group 'Group Test' to the Windows 'Desktops - All types'.

```
AddHostsToGroups ("GROUP_NAME='Group Test'",
"GROUP_NAME='Desktops - All types' OS_TYPE='WINDOWS'
VERSION='5.0 r173'")
```

If a host to be added has an operating system different from that of a group matching the group object expression, then the host will not be moved into the group. This will not generate an error condition – the host will simply be skipped.

```
Boolean RemoveHostsFromGroups (String hostExp, String
groupExp)
```

This function removes ALL the hosts matched by the host expression from ALL the groups matched by the group expression. This function does not add the hosts to any groups. It just removes.

The following example removes all hosts in the group 'Group Test' from the Windows 'Desktops - All types'.

```
RemoveHostsFromGroups ("GROUP_NAME='Group Test'",
"GROUP_NAME='Desktops - All types' OS_TYPE='WINDOWS'
VERSION='5.0 r173'")
```

```
Boolean MoveHosts (String hostExp, String fromGroupExp,
String toGroupExp)
```

This function removes ALL the hosts matched by the host expression from ALL the groups matched by the "from group expression" and adds these hosts to ALL the groups matched by the "to group expression".

It is worthwhile to note that `MoveHosts` could theoretically have different results from running `AddHostsToGroups` and then `RemoveHostsFromGroups`. The reason for this is that `MoveHosts` calculates the "from" and "to" groups *before* doing any adding, whereas running the latter combination will cause the `Remove`'s "from" to be calculated *after* hosts have already been added. In almost all practical cases, however, the two approaches will be identical.

The following example moves all hosts from the group 'Group Test' to the Windows 'Desktops - All types'.

```
MoveHosts ("GROUP_NAME='Group Test'",  
"GROUP_NAME='Desktops - All types' OS_TYPE='WINDOWS'  
VERSION='5.0 r173'")
```

This function will fail if the “from group” matches groups but the “to group” does not. This is because this would result in the deletion of hosts from the “from group”, and this is probably not what the caller had intended. This function will also fail if one or more hosts cannot be moved due to an OS incompatibility.

```
Boolean DeleteHosts (String hostExpression)
```

This function sends all hosts that match the host expression to the CSA MC Recycle Bin. Obviously, this function should be used with caution.

Manipulating Hosts

The following functions manipulate hosts in some way. All these functions take as a parameter a host expression (object expression).

Note that for all these functions, it is not considered an error/failure if the host expression returns no hosts. The reasoning for this is that scripts may simply want to manipulate certain hosts that meet certain criteria, but it may be common for no hosts to meet the object expression criteria at any given time.

If a script wants to detect the case in which no hosts were manipulated because no hosts matched the hosts expression, the script can call the `GetStatus` function.

```
Boolean SendHint (String hostExp)
```

This function sends a hint to all hosts that match the host expression.

This function does not block until all the hosts receive hints. Rather, the scripting interface infrastructure simply places “tags” in the CSA MC database that tell the CSA MC hint infrastructure to give priority to the appropriate hosts. Depending upon a number of factors, including the speed of the processor, the number of hosts to be hinted, etc., there may be some delay before the hints are sent. Bear in mind this delay exists to ensure that the polling server is not overwhelmed by polls.

This function returns true on success and false on failure.

```
Boolean ResetAgent (String hostExp, Integer resetMask)
```

This function resets the Cisco Security Agent. It works in a manner similar to the corresponding function in the CSA MC UI. When the function is run, the scripting interface infrastructure sets a flag in the CSA MC database that tells the CSA MC to reset the appropriate agents the next time these agents poll.

Callers of this function should calculate the reset mask by adding the desired values from the list below:

- 1 – Cached Responses and Logging
- 2 – Local Firewall Settings
- 4 – Learned Information
- 8 – System Security
- 16 – System State
- 32 – Untrusted Applications
- 64 – User Query Responses

For example, to reset the “System Security” and “System State” only, the mask should be 24, which is the addition of 8 and 16.

Important Note: This function blocks until the CSA MC database flags have been set, and then it returns true on success and false on failure. Note that the return of this function does not mean that all the agents have been reset – it simply means that the CSA MC has been configured to reset the agents the next time they poll, whenever that may be. Note that you may be able to expedite the polling of the agents by calling the `SendHint` function after calling the `ResetAgent` function.

Getting Host Information

The following functions return information about the specified host(s). For each of these functions, an empty string array is returned either on error or if there are no matching hosts. `GetStatus` will return 0 on success, 1 if the host expression yields no hosts, or another number for other kinds of failure.

```
StringArray GetHostInfo (String hostExpression)
```

Each string in the returned array contains the information from the Hosts page on the CSA MC system for a given host.

```
StringArray GetLastIpAddrs (String hostExpression)
```

This function can be used to get the IP address history of one or more hosts. The format of each string in the array is as follows:

```
<ip addr> <hostname> <ACTIVE | INACTIVE>
```

For example:

```
1.1.1.1 host1 ACTIVE
1.1.1.2 host2 ACTIVE
10.10.10.10 host3 ACTIVE
10.10.10.11 host4 INACTIVE
10.10.10.12 host5 INACTIVE
```

```
StringArray GetDiagnosticInfo (String hostExpression)
```

This function can be used to get diagnostic information from the hosts that match the host expression argument.

Because of the way diagnostic information is collected and stored by the CSA MC server, this function behaves a little differently from other CSAAPI functions that get host information.

When the `GetDiagnosticInfo` function is run, the CSA MC system does two things.

1. The server sends back to your script whatever diagnostic information exists currently in the database for the appropriate hosts. If there is no existing diagnostic information in the database for a particular host, then for that host your client script will receive a string asking you to check back later for updated diagnostic information.
2. The server asks the Cisco Security Agent(s) on the appropriate host(s) to send new diagnostic information to the CSA MC system. Each host will send new diagnostic information to the CSA MC system the next time it polls (default interval is ten minutes) and this information will be stored in the CSA MC database.

Therefore, in order to get the latest diagnostic information for a particular host, it is recommended that you call the `GetDiagnosticInfo` function *twice*. This first call is made simply to ask the host for the latest information – the return value can simply be discarded. The second call is made after either calling `SendHint`

or waiting for the polling interval to elapse. The returned string array from the second call will represent the latest diagnostic information available from the host.

Getting Overall System Information

The following functions return information about the overall system.

```
StringArray GetMostActive (String objType, Integer
numOfObjects, String timespan)
```

This function returns the ‘n’ most active objects over the specified time span. This function provides the same information provided on the Status Summary page of the CSA MC UI.

Important Note: The only two object types currently supported by this function are `HOST` and `RULE`.

Important Note: The only legal values for the “timespan” variable are: `DAY`, `WEEK`, and `ALL`.

For hosts, the returned array will be a list of strings with a format like this:

```
myhost [W] - 174 events
```

For rules, the returned array will be a list of strings with a format like this:

```
File access control [id=266], CSA MC Security Module [W,
V5.2 r0] - 6 events
```

Examples:

To get a list of the 10 most active hosts over the past 24 hours, use the syntax:

```
GetMostActive ("HOST", 10, "DAY")
```

To get a list of the 20 most active rules over the past week, use the syntax:

```
GetMostActive ("RULE", 20, "WEEK")
```

```
Integer GetLastDayEventCount ()
```

This function returns the number of events generated in the past 24 hours.

```
StringArray GetReportNames ()
```

This function returns a list containing the names of all currently defined reports.

```
StringArray GetHostTaskNames ()
```

This function returns a list containing the names of all currently defined Host Management Tasks.

Getting Event Information

The following functions return information about events.

```
StringArray GetLatestEvents (String objType, String  
objExpression, Integer numEvents)
```

This function returns the latest events corresponding to the objects matching the expression.

Important Note: This function does not support all the defined object types. The following types are supported: HOST, GROUP, POLICY, RULE_MODULE, RULE. The following types are NOT supported: CLASS, VAR.

```
StringArray GetLatestAudit (Integer numEvents)
```

This function returns last 'n' audit trail events.

Getting Reports

```
Integer GenerateReport (String reportName, String  
emailAddress)
```

This function generates a report. If an *empty* email address is provided, then the report is simply placed in the default report directory on the CSA MC. If, on the other hand, a legitimate email address is provided, then the report is emailed to the appropriate address and is removed from the default report directory on the CSA MC.

This function does not block until the report is created. Instead, this function creates a process that creates the report. This function returns a Process ID that can be used with `GetStatus` to determine when/whether the report was successfully generated.

Note: In order for this function to send email, the scripting interface must be informed of the name of the server to which to send emails. To do this, use the `SetConfigVar` function to set the "csaapi.email_server" parameter to the name of the email server. Note that this must be done only once.

Note: To allow the scripting interface to send email, you must first make a minor modification to the CSA MC rule that governs permissible web server actions. Please see the scripting interface sample script README.txt files for details.

**Note**

To email a report to multiple addresses, separate the email addresses with a semicolon (;). For example: joe@company.com;jane@company.com
