Quest®

Quest® InTrust 11.3

Customization Kit

If you have any questions regarding your potential use of this material, contact:

Quest Software Inc.

Attn: LEGAL Dept

4 Polaris Way

Aliso Viejo, CA 92656

Refer to our Web site (https://www.quest.com) for regional and international office information.

Patents

Quest Software is proud of our advanced technology. Patents and pending patents may apply to this product. For the most current information about applicable patents for this product, please visit our website at https://www.quest.com/legal.

Trademarks

Quest, the Quest logo, and Join the Innovation are trademarks and registered trademarks of Quest Software Inc. For a complete list of Quest marks, visit https://www.quest.com/legal/trademark-information.aspx. All other trademarks and registered trademarks are property of their respective owners.

Legend

CAUTION: A CAUTION icon indicates potential damage to hardware or loss of data if instructions are not followed.

IMPORTANT, **NOTE**, **TIP**, **MOBILE**, or **VIDEO**: An information icon indicates supporting information.

InTrust Customization Kit
Updated - May 2017
Version - 11.3

Contents

InTrust Customization Overview

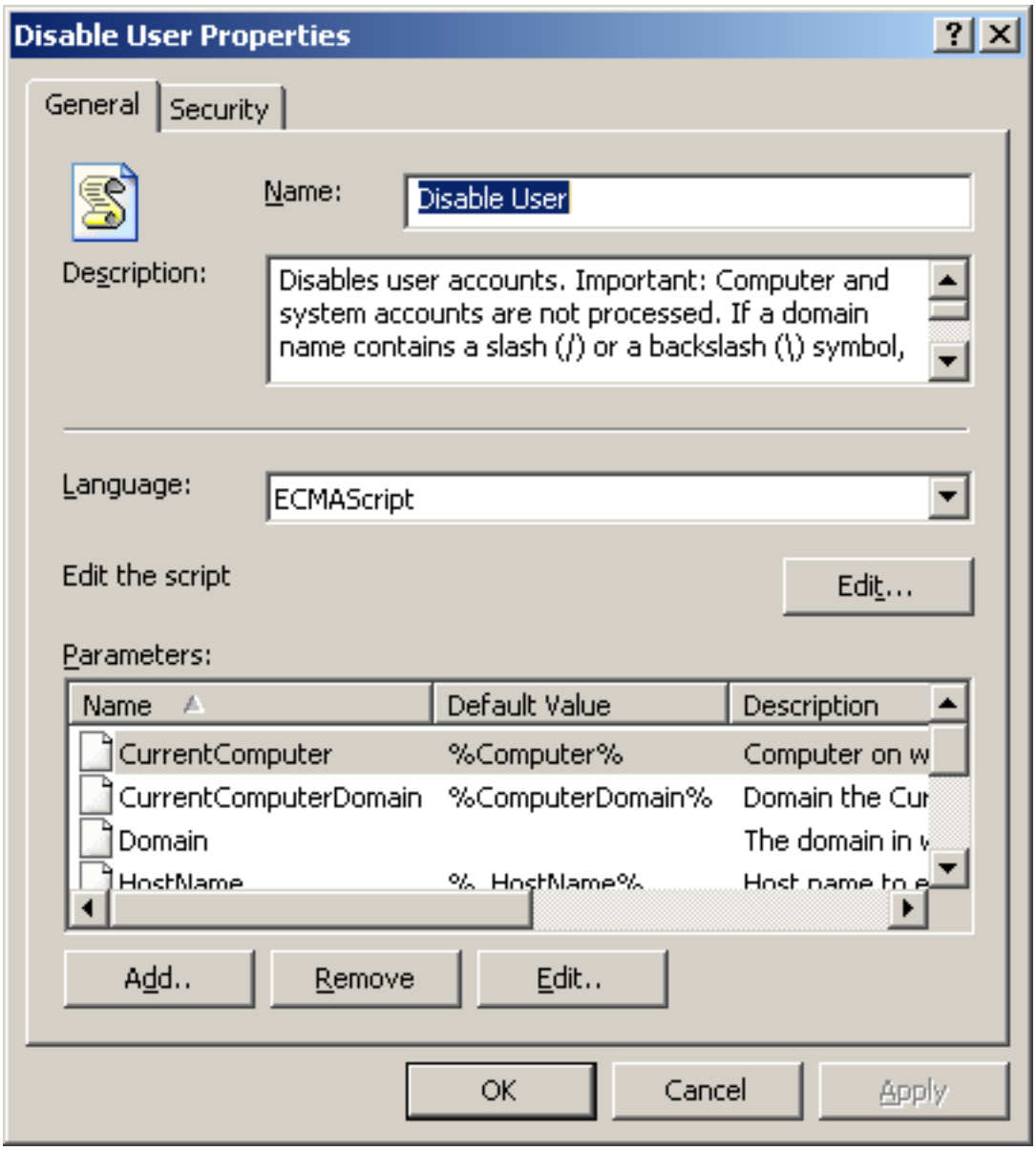
By using scripts and Quest InTrust as the framework to execute them, you can enhance the InTrust toolset. Scripting extends the following areas of InTrust functionality:

- Creation of real-time monitoring rules
- Creation of scripted data sources (primarily for real-time monitoring)
- Fine-tuning data sources for custom event logs in text format (for auditing purposes)
- Editing filters for gathering, consolidation and import policies
- Configuring custom response actions for rules
- Enumeration of sites

InTrust Script Objects

InTrust script objects provide logic and automation facilities accessible from different parts of InTrust: response actions in real-time monitoring rules, advanced site enumeration algorithms, and so on. In InTrust Manager, script objects are located in **Configuration | Advanced | Scripts**, and contain the following:

- Parameters used in the body of the script but defined outside it
- Actual script code



Parameters

Parameters are variables that the script exposes to its callers. For example, if a script is a response action designed to disable a user account, it must get the user name as a parameter.

Parameters are specified in the properties dialog box of the script object. The parameter list must include all parameters that the script expects to get from the rule. Use the **Add**, **Remove** and **Edit** buttons to work with the parameter list. For details about supported parameter types, see Customizable Parameters.

Script Body

To supply or edit the script code, open the properties of the script object you need and click the **Edit** button with the "Edit the script" label next to it. Use the scripting language selected from the **Language** list box.

When you need to use a parameter in the script, call the **Parameters** method of the **ScriptContext** object and pass the parameter name as the argument. For example, the script above has the "CurrentComputer" parameter, defined in ECMAScript as follows:

```
var strComputerName = Parameters["CurrentComputer"];
```

in JavaScript you must use the following call to get the current computer name:

```
ScriptContext.Parameters("CurrentComputer")
```

As long as the "CurrentComputer" parameter is included in the parameter list of the script object, this is a valid call.

Customizable Parameters

Script parameters can be exposed by several types of InTrust objects: rules, script objects, and data source filters. This topic describes the supported data types for parameters.

AccessType

Lets you specify access types used by events in Windows logs. In the user interface, this type is represented by a check box list.

Example: "*%%1023*", "*%%1111*", "*%%2222*"

Here, numbers correspond to access type IDs.

Text

Example: Some text.

Number

Example: 512.

List

Comma-separated list. Example: "dog", "cat", "bird". Commas and quotation marks cannot be used inside list elements.

DateTimeRange

This type has the following format:

"yyyy/mm/dd hh:mm:ss"

The number of digits in a field is not fixed. You do not have to insert leading zeros. You can omit either the date part (yyyy/mm/dd) or the time part (hh:mm:ss) but not both at once. If you omit the date, do not leave a leading white space.

The mm (month), dd (day), mm (minute) and ss (second) fields can be omitted. In this case, they are assumed to be 0.

Example:

"0/0 1:0" is the same as "0/0 1" or "1", meaning one hour.

If you omit the entire date, the minutes and the seconds, the specified number is assumed to be a date. For example, if only the number 1 is specified, it is treated as a year, although you might expect it to mean an hour.

To determine which part is omitted, look at the separator characters (/ or :).

ExpectedTime

Time at which an event is expected in a "missing event" rule. Specified in the cron format, meaning five numbers separated by spaces or tabs. The order is as follows:

- minute (0-59),
- hour (0-23),
- day of the month (1-31),
- month of the year (1-12),
- day of the week (0-6 with 0=Sunday).

Example: "0 1 * * 4,6", "1:00". Here, the event is expected every Wednesday and Saturday at 1 AM.

RangeList

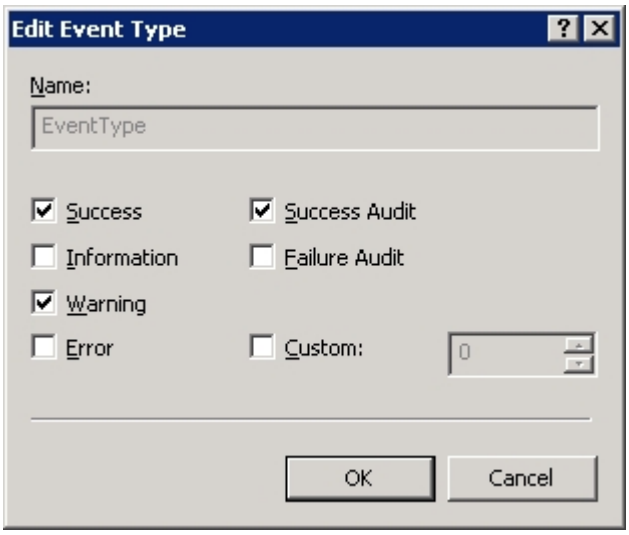
This type has the following format:

"min_1-max_1,min_2-max_2,...,min_N-max_N"

Here, **min_x** is the minimum value in a range; **max_x** is the maximum value. The maximum value and its leading hyphen are optional. Both the minimum and the maximum are non-negative integers.

EventType

This type is a list of values enclosed in quotation marks for the EventType field in Windows logs. The selection dialog box for this parameter is shown in the screenshot:



This selection corresponds to the value "0,2,8".

In the XML markup for REL-based rules, this type can be used as in the following example:

```
in_range(EventType, <parameter name="MyEventTypes"/>)
```

Filter

This type of argument is a container for an expression, which can optionally have its own arguments.

The referenced arguments are enclosed in the **<filter>** tag pair, as follows:

```
<argument name="My Event Filter" class="Filter">
  <value>
    <filter type="EventFilter" version="1.0">
      <arguments>
        ...
      </arguments>
    </filter>
  </value>
</argument>
```

For a usage example, create a rule for a Windows event log or Syslog based on the "Single event" template and using one of the predefined filters. View the resulting markup.

Choice

This type lets you select one or more values from a list. Use the "List" type to represent the choice in the user interface, as follows: type = "List".

Usage example:

```
<argument usedefault="true" name="sample_arg" class="Choice" type="List">
  <choices>
    "arg_value1", "arg_value2", "arg_value3", "arg_value4"
  </choices>
  <value>
    "arg_value2"
  </value>
  <default description="descr">
    "arg_value1", "arg_value2"
  </default>
</argument>
```

The **<choices>** tag pair stores the list of possible values that you can select from.

```
<filter type="EventFilter" version="1.0">
 <arguments>
    <argument usedefault="false" name="sample_arg" description="" class="guid">
      <value>
        ...
      </value>
      <default description="descr">
        Sample description
      </default>
    </argument>
```

```
  </arguments>
  <body>
    ...
    <parameter name="sample_arg"/>
    ...
  </body>
</filter>
```

Password

This type lets you securely specify a password for an authentication operation. Do not specify the password directly in the XML code. The implementation of this parameter type permits password editing only in a graphical prompt.

```
<argument displayname="password" name="password" description="Password"
class="Password">
  <value/>
</argument>
```

InTrust Server Tracing

InTrust Server provides tracing capabilities in most of its components. Tracing is enabled and disabled for particular components in the **adctracer.ini** file.

For InTrust Server, the location of this file is **Server\ADC\adctracer.ini** in the folder where InTrust Server is installed. For InTrust agents, the location is **ADCAgent\adctracer.ini** on the processed computer.

The first line of the file specifies the location of the traces. The default locations are as follows:

- InTrust Server traces are written to **Server\ADC\tracing** in the folder where InTrust Server is installed.
- InTrust agent traces are written to **ADCAgent\tracing** on the processed computer.

Configuration File Format

The entries in the **adctracer.ini** file use the following format:

```
ComponentName=Number
```

Here, **ComponentName** is the name of the InTrust component for which you want to see traces; **Number** is the tracing level. Example:

```
MSNNSiteProvider=40
RELMatcher=40
```

By default, all entries are commented out with the number sign (#). Uncomment those entries for which you need traces.

The **adctracer.ini** file sets the default tracing level for each component. For the trace to be recorded, the tracing level in the trace-writing function must be less than the value specified in **adctracer.ini**. For example, if you specify level 50 in a tracing function in your site enumeration script, and the **MSNNSiteProvider** entry is set to the default value of 40, your trace will not be recorded.

Distributing Files to Agent Computers

You can use the file distribution mechanism in InTrust rules to copy files to agent computers. Take the following steps:

1. Create a site that includes the computers you need.
2. In **Configuration | Advanced | Scripts**, create a script object that defines the **OnInstall** and **OnUninstall** functions. Example:

```
function OnInstall()
{
var objEnv = new ADCEnvironment();
var srcpath = objEnv.ExpandEnvironmentString("%ADC_INSTALL_
PATH%\\data\\dda\\%adc_org_id%");
<where_to_put_the_file>";
var fso = new ActiveXObject("Scripting.FileSystemObject");
<file_name>",destpath+"\\<file_name>",1);
}
function OnUnInstall()
{
}
```

   The **<file_name>** placeholder stands for the name of the file you want to distribute. All distributed files first arrive in the **%ADC_INSTALL_PATH%\data\dda\%adc_org_id%** folder, where:

   - **%ADC_INSTALL_PATH%** is a local environment variable storing the agent installation folder
   - **%adc_org_id%** is an InTrust organization parameter specifying the ID of the current InTrust organization

3. In InTrust Manager, go to **Configuration | Advanced | Distributable Files** and add the file you need to distribute. In the properties of the file object, select the **Run this script upon module delivery to the agent side** option, and select the script from the previous step.
4. Create a rule based on any Windows log data source, and add to it the file from the previous step as a distributable module. Enable the rule.
5. Create a real-time monitoring policy that applies your rule to the computers you want to copy the file to. Activate the policy.

You can adapt this procedure to your specific needs: for example, add actions besides file copying or parameterize the file destination path, and so on.

Case Study: Enabling Tracing on Multiple Computers

Objective

Automatically enable tracing on specific computers.

Solution

Distribute the **adctracer.ini** file (see InTrust Server Tracing), which has been edited to enable tracing.

Details

This procedure is based on the generic steps described above, so refer to them for more information.

1. Make a copy of an existing **adctracer.ini** file, and edit the parameters in it as necessary.
2. Create a site with the computers you need.
3. Create a script object with the following code:

```
function OnInstall()
{
var objEnv = new ADCEnvironment();
var srcpath = objEnv.ExpandEnvironmentString("%ADC_INSTALL_PATH%\\data\\dda\\
{42D329C8-7150-485B-90F1-8FA1D224A767}");
Trace(40, "Source Path: " + srcpath);
var destpath = objEnv.ExpandEnvironmentString("%ADC_INSTALL_PATH%");
Trace(40, "Destination Path: " + destpath);
var fso = new ActiveXObject("Scripting.FileSystemObject");
Trace(40, "Copying " + srcpath+"\\adctracer.ini"+" to
"+destpath+"\\adctracer.ini");
fso.CopyFile(srcpath+"\\adctracer.ini",destpath+"\\adctracer.ini",1);
}
function OnUnInstall()
{
}
```

4. Add your copy of **adctracer.ini** as a distributable file, and associate your script with it.
5. Create and enable a rule that provides the file.
6. Create and activate a real-time monitoring policy that specifies the computers you need.

How to...

Working with Data Sources

Data sources are InTrust's representations of the event logs it works with. One of the InTrust data source types is the Script Event Provider.

This data source is actually a scripting component that InTrust executes periodically for auditing and real-time monitoring purposes. Scripts are meant to return one or more event records with filled-in fields. However, the initial information that the scripts get does not exist in event format.

The format your script works with is up to you. For example, the script can analyze text files.

Topics

See the following topics for details:

Creating a Data Source

If you want to create your custom data source from scratch, InTrust provides two starting points for convenience, depending on what you want the data source to do:

- Analyze text logs or other text files
- Perform arbitrary tasks that go beyond text file analysis

Text Log Data Sources

File-processing scripts are state-based. Such a script checks whether a file is present, or periodically parses a file and reconstructs events from the file changes it detects.

One of the data source types is the general-purpose custom text log data source. It is implemented as a script that processes specified files for auditing purposes.

You cannot specify the desired script directly. You must first create an outline for the data source in either Basic or Advanced mode. For more information about text log data source creation modes, see Auditing Custom Logs.

Data sources completed in Raw mode give you the advantage of easy flow control. Unlike Advanced mode, you do not have to rely on consecutive regular expressions and their order. You can introduce conditional jumps and eliminate regular expressions altogether. This makes Raw mode more suitable for many situations, including markup parsing.

Automatic data source creation gives you a starting point and spares you the effort of outlining the script structure manually. After you have created the initial data source, do the following:

1. Open the properties of the data source.
2. On the Settings tab, click **Convert** to and select **Raw**.
3. Edit the resulting script using the code editor.

Custom Scripted Data Sources

InTrust requires the functionality of the Windows Script Host (WSH) object model on all computers that it monitors. If Windows Script Host is installed on a monitored computer, its objects can be used by InTrust.

If WSH is not installed (for example, on a Linux computer), then the scripting component can provide basic WSH functionality (available through ECMAScript) and ensure that InTrust scripts have access to the necessary objects.

The object model is defined by the scripting language you select in the properties of the data source.

To create a scripted data source

1. Expand the **Configuration** node in the InTrust Manager snap-in.
2. Right-click **Data Sources** and select **New Data Source** to start the New Data Source wizard.
3. Select the **Script Event Provider** type and complete the wizard.

The wizard prompts you for the following information:

- Scripting language
  You can write your script in JScript, VBScript or ECMAScript. The choice of language is up to you and should depend on the platform and available objects. You can use JScript or VBScript if the script is going to work on Windows computers. However, if the data source represents a log on a platform such as Linux, you should use ECMAScript, which is a cross-platform language.
- The script itself
  Supply a script that meets the requirements described earlier.
- How often the script executes
  Frequent launches are suitable for real-time monitoring uses. For data collection, the script does not have to run very frequently.
- Script parameters
  Parameters are values that can be set externally without modifying the script. They are set in the user interface, and the values are stored in script variables. Script parameters are meant for easy access to the script's configurable portions.

Generating Events

To generate an event for InTrust to process, a data source script should do the following:

1. Create one or more event structures using the **CreateEvent()** function.
2. Fill in the fields of the events based on data analysis results.
3. Pass the events on with the **Submit()** method.

The **CreateEvent()** function does not have any arguments. Use the function as follows:

```
var myEvent = CreateEvent();
//myEvent now stores the event structure
```

After making the event accessible through a variable, you can fill in the event fields with values. The fields can be either predefined or created "on the fly". By convention, the names of some predefined fields start with underscore characters (_). Such fields should not be edited.

The predefined event fields are listed in Event Fields.

The following is an ECMAScript example of field-processing operation:

```
myEvent["MyField"] = "myValue";
// Creates the MyField field
// and writes the value "myValue" to it
//
myEvent.MyField = "myValue";
// Does the same as the previous example
```

In Windows Script Host languages, the syntax for setting field values differs. Here is a JScript example:

```
myEvent.Values("MyField") = "myValue";
// Creates and fills in a custom field
```

Before you submit an event from a script in ECMAScript, make sure the **TimeWritten** and **TimeGenerated** fields are set.

Like **CreateEvent()**, the **Submit()** method does not take any arguments. Use it as in the following example:

```
myEvent.Submit();
```

The events that scripted data sources submit are passed on to real-time monitoring rules or cached.

Typical Custom Text Log Data Source Logic

Glossary

The audit script host provides access to scripting objects specific to InTrust auditing. The terms audit script host and **GenericScriptProvider** can be used interchangeably.

The audit script engine is the intermediary between the audit script host and the computer where the script is running.

The audit script object model is exposed by the script engine to the scripts.

The audit script position processor is the object that executes the position comparer script.

Typical Usage

The following logic is typically used in custom text log data sources:

1. The audit script host creates an audit script host instance initialized by a script.
2. The audit script host makes an **Audit_EnumInstances** call from the script, and uses a computer name as a parameter. The computer can be represented by a NetBIOS name, an IP address, and so on.
   The call returns a two-dimensional array, in which the "parent" array has only one element, and the nested array has two elements. The first of these two elements is the same computer name that was passed to **Audit_EnumInstances**. The second element is the display name of that computer, which is then passed to **Audit_Connect**. For Windows systems, this is the NetBIOS name; for UNIX systems it is the local name of the host. The display name is exposed in the session list of the InTrust Manager snap-in.
3. The audit script host makes an **Audit_BeforeCollection** call, which can perform all kinds of preparatory activity before gathering.
4. The audit script host makes an **Audit_Connect** call with an instance parameter. The instance parameter is the value previously returned by **Audit_EnumInstances**. The log parameter corresponds to the EventLog field in an InTrust data store.
5. The audit script host makes an **Audit_Seek** call, which expects a **Position** object. The script can use the **Position** object to gather audit trails in increments. The object stores information required by **Audit_Seek** to find the place in the audit trail where gathering stopped the last time. If the precise position is found, true is returned. Otherwise, the result is false. If false is returned, the details of the corresponding session state that the position could not be found.
6. The audit script host makes an **Audit_CollectEvents** call. This is the stage at which gathering takes place. During the gathering, the script must call the **AuditProvider.SubmitEventPositionPair** method. This method returns pairs each composed of an **Event** object and a corresponding **Position** object.
   For the custom text log data source to work properly, each **Position** object must contain an element with the following index in the values array:
   {3C2E0E29-790F-47bf-99B2-8F71DD23FA07}
   The element with this index must contain the code of the function that compares positions using **Audit_ComparePositions**. The audit script position processor uses this code for comparison. The text must be identical in both positions; otherwise comparison will produce an error.
   This is required so that the custom script can implement a comparison function and positions can be compared without the custom script.
   To see an example of how the {3C2E0E29-790F-47bf-99B2-8F71DD23FA07} index is used, open the properties "Custom Text Log data source template" script and click **Edit**. Locate the definition of the **PosHelper** function and note the way the index is handled.
7. Gathering finishes after **Audit_CollectEvents** completes.
8. The audit script host makes an **Audit_AfterCollection** call. If the success parameter is true, it means that events submitted by the script have been successfully stored. If success is false, this indicates that event storing has failed. For example, the script can use this parameter to clear a previously gathered log. However, you may lose some events in this case.

The script can use the following methods to output diagnostic information:

- AuditProvider.Trace to write to the AdcEventDataManager trace
- AuditProvider.LogMessage to write to session details

- AuditProvider.OperationStatus to write to session details

For details about the InTrust object model, see Object Library.

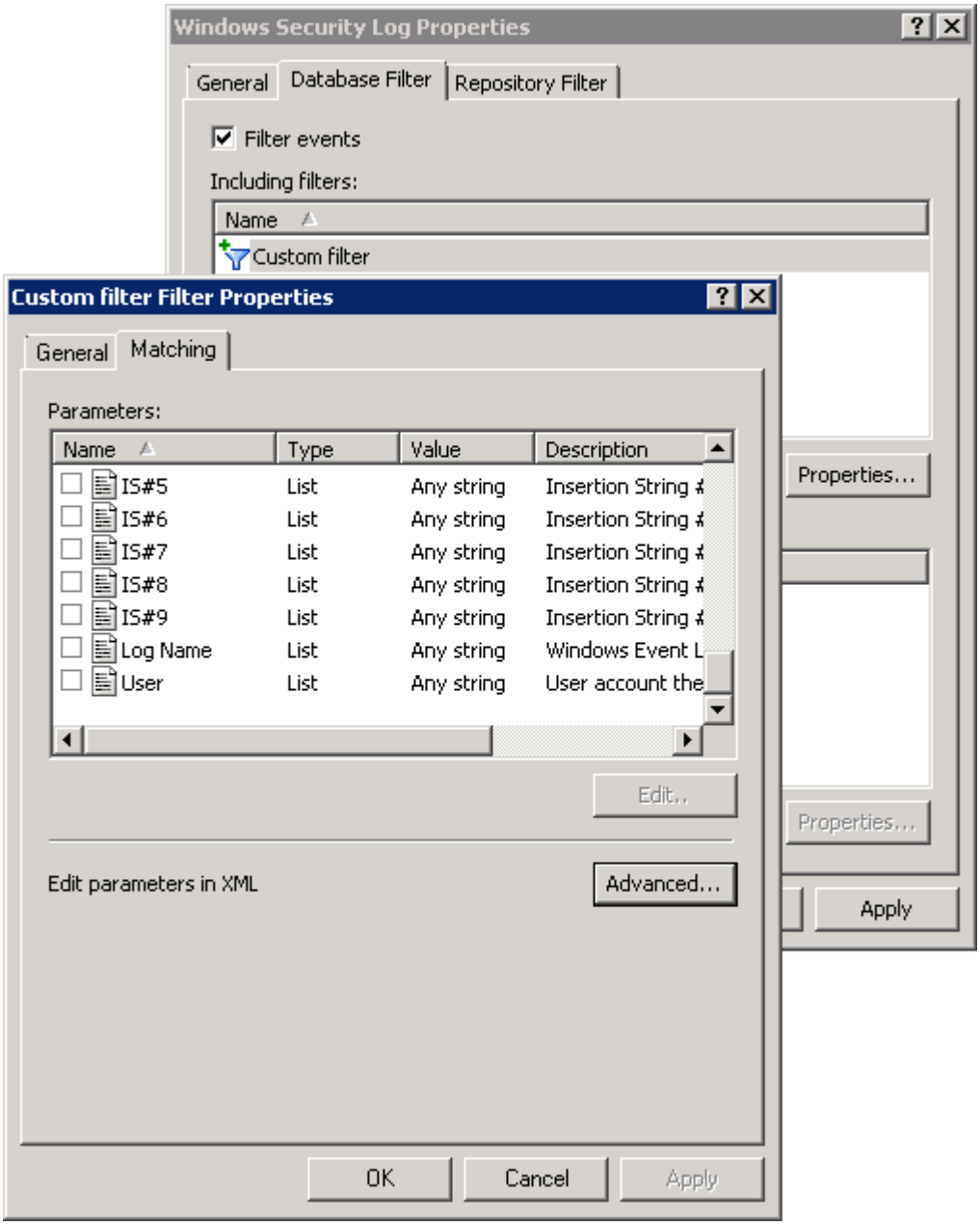
Customizing Data Source Filters

Filters on data sources are defined for gathering, consolidation and import policies; these filters can be customized using REL.

The filters are available for editing in the properties of a data source. You can open the properties of an existing data source directly in InTrust Manager. Alternatively, you can configure the filters as you create a new policy.

NOTE: Renaming and editing the filter instance that you have added to a data source does not affect the predefined list of available filters. Your modified filter is stored with its respective data source.

To modify the parameters and matching conditions, click the **Advanced** button in the properties of the filter.



The following is an example of a simple filter:

```
<filter type="EventFilter" version="1.0">
  <arguments>
    <argument displayname="Computer List" name="Computer List" description=""
class="List">
      <value />
    </argument>
  </arguments>
  <body>
    _DataSourceName="Security" and in(_HostName, "wi", array(<parameter name="Computer
List">))
```

```
  </body>
</filter>
```

In this example, the filter accepts events only from a specific (case-insensitive, wildcard-enabled) list of computers.

For details about defining parameters, see Customizable Parameters. For a list of predefined event fields that you can filter by, see Event Fields.

Event Fields

The following table lists predefined fields in event records. When you create an event, do not modify any of the **bolded** fields.

FIELD	DESCRIPTION
RecordNumber	Number of the record in the event log, used for storing the position of the last gathered event.
TimeGenerated	Time when the event was generated.
TimeWritten	Time when the event was written to the log.
EventID	ID of the event in the InTrust gathering session.
EventType	Type of the event.
NumericCategory	Integer representation of the event category.
StringCategory	String representation of the event category.
Source	Name of the event source.
Computer	Computer on which the event occurred.
ComputerDomain	Domain of the computer on which the event occurred.
UserBinarySid	SID of the user who produced the event.
UserName	Name of the user who produced the event.
UserDomain	Domain of the user who produced the event.
Description	Description of the event.
EventData	Binary data of the event.
VersionMajor	Major operating system version number of the computer on which the event occurred. For example, the major version of Windows XP is 5.
VersionMinor	Minor operating system version number of the computer on which the event occurred. For example, the minor version of Windows XP is 1.
PlatformID	Platform (operating system) ID of the computer on which the event occurred.
AccountName	Name of the user who produced the event, in **domain\user** format.

FIELD	DESCRIPTION
TimeLocal	Time when the event was written to the log; this time is local to the computer where the event was logged.
_ID	Unique identifier of the event (a string that represents the object's GUID)
_LocalTime	Local time of event generation.
_GMT	GMT time of event generation.
_Priority	Event priority. This field is not used, and is always set to 2, meaning Normal.
_DataSourceName	Name of the log.
_ProviderName	Name of the InTrust data source.
_DataSourceId	GUID of the log.
_HostName	Name of the host where the event was generated.

Creating Rules

An InTrust real-time monitoring rule is a logical structure that is used to constantly analyze event flow, which is created by the InTrust agent on the monitored computer. As a result of the analysis, the rule either triggers further actions or does nothing.

If the rule detects the condition that it is designed for, this means that the rule is matched. In this case, possible actions are as follows:

- Sending alerts
- Running a program
- Running a script
- Launching an InTrust task
- Setting an audit policy
- Sending an SNMP trap

Every rule is implemented in XML with embedded scripts. The XML code for a rule has three distinct sections:

1. Parameter descriptions
   This section describes rule parameters. Rule parameters are displayed in the user interface so that you can set their values with interactive controls. They are stored as variables for use in the matching script. The terms "parameter" and "argument" can be used interchangeably.
2. Optional pre-filtering script
   This script filters events for matching. It minimizes the number of events the rule has to process by filtering out anything that the matching process does not have to consider. The script helps reduce bandwidth and optimize performance.
3. Matching script
   This script processes pre-filtered data and either returns event records or does nothing depending on

input. Event records are passed on to the InTrust framework and can be used in real-time monitoring alerts. Returning event records means that the matching process was successful, and InTrust takes all actions associated with a matched rule.

For details about the structure of a rule, see Rule Structure.

To create a real-time monitoring rule

1. Expand **Real-Time Monitoring | Rules**.
2. Right-click the necessary rule group and select **New Rule**.
3. In the New Rule Wizard that starts, select a data source type.
4. Select one of the predefined rule templates or the **Custom Rule** option.
5. Complete the wizard.

There are three ways to create a rule with the wizard, as follows:

1. Using a predefined template
2. Using a custom REL-based template
3. Using a custom ECMAScript-based template

Predefined templates cover most real-time monitoring situations. In addition, if a predefined template does something similar to what you need, but you want the rule's behavior to differ slightly, you can use the predefined template and edit the resulting rule later.

REL (rule expression language) is a built-in language designed specially for creating rules. For example, rules created using predefined templates are XML structures with embedded REL code.

REL and ECMAScript are both used for the same purpose, but have very different ideologies. The following table shows the pros and cons of each rule creation method.

METHOD	ADVANTAGES	DISADVANTAGES
Predefined templates	• Easy, quick and intuitive • Does not require scripting skills • Suitable for most common monitoring needs	• Provides a limited set of data filtering options and patterns • Cannot create custom matching logic
Custom REL-based templates	• Lets you define custom algorithms for pre-filtering and matching • Ideal for moderately complex solutions where predefined data filtering options are not applicable	• Has a learning curve • Defines a rigid logical structure where conditional behavior is difficult to implement
Custom ECMAScript-based templates	• Powerful and flexible scripting environment • Achieves tasks of any complexity • Best for detecting event patterns	• Can be impractical for moderately complex solutions • Can be slower than a an equivalent REL-based rule

For the matching script, it makes a difference where matching occurs—on the InTrust server side or on the agent side.

Matching on the server side enables you to use the local resources of the InTrust server, such as libraries and applications. Matching on the agent side lets you use the resources of the target computer and get additional information associated with collected data, if necessary. In either case, shared network resources are available to rules.

The following are examples of where server-side and agent-side rules can be useful:

- Server side: depending on what events come in from a certain computer, you can look for related events on any other monitored computer to create complex alerts.
- Agent side: you can easily create a rule for members of computer local administrative groups on a target computer.

Importing and Exporting Rules

You may want to make only minor changes to an existing rule. You may also want to use your own text file-based rule template instead of the custom rule wizard. In these situations, the import and export features of InTrust come in handy.

To export a rule to an XML file for further customization, right-click the rule in InTrust Manager and select **Export**.

To import a customized rule, use the **InTrustPDOImport** command-line utility, which is located in the ***<InTrust_installation_folder>*\InTrust\Server\ADC\SupportTools** folder on the InTrust Server computer.

See Also

Rule Structure

Examples

Rule Structure

A real-time monitoring rule has the following XML structure:

```
<?xml version="1.0" encoding="utf-8" ?>
 <rule type="REL" version="1.0" language="jscript">
 <arguments>
 <argument usedefault="true" name="sample_arg" description="" class="Text">
 <value>sample_arg_value</value>
 <default description="descr">
 sample default value
 </default>
 </argument>
 ...
 </arguments>
 <prefilter>
 ...
 </prefilter>
 <body>
 ...
 <parameter name="sample_arg"/>
 ...
```

```
</body>
</rule>
```

The **<rule>** tag pair encloses the entire rule and has the following attributes:

- **type**
  This attribute should always be "REL".
- **version**
  This attribute specifies the version of the scripting language used in the rule. For both languages, the current version is 1.0.
- **language**
  This attribute set whether or not the rule is ECMAScript-based or not. For ECMAScript-based rules it should be "jscript".

The **<rule>** tag pair encloses the following:

- **<arguments>** tag pair
- optional **<prefilter>** tag pair
- **<body>** tag pair

Specifying Arguments

The **<arguments>** tag pair contains definitions of rule arguments and sets their default values. Inside the pair, each argument is represented by an **<argument>** tag pair with the following attributes:

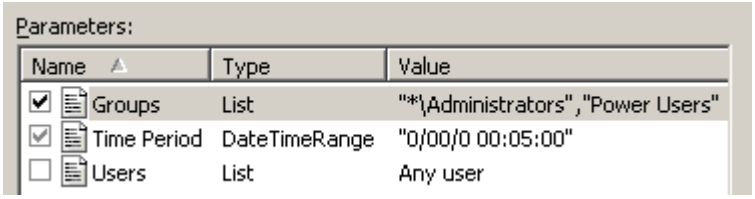
- **name**
  The name of the argument. Use this name in the rule script to get the argument value.
- **description**
  Description of the argument. This attribute is optional. If specified, the description is displayed in the user interface.
- **class**
  Class, or data type, of the argument. This is a mandatory attribute. Supported classes are listed further on.
- **usedefault**
  Binary attribute specifying where the rule gets the argument value. If true, the value specified in the **<default>** tag pair is used. Otherwise, the value from the **<value>** tag pair is used. This attribute is optional. If you do not supply it, the value from the **<value>** tag pair is used.

Inside the **<argument>** tag pair, specify the following tag pairs:

- **<value>**
  The current value of the argument.
- **<default>**
  The value that is used if the argument is left unchanged. The **<default>** tag has an optional description attribute. Use this attribute to make the default value descriptive in the user interface.

Compare the following snippet of code and the way the user interface represents it in the rule properties on the **Matching** tab:

```
...
 <arguments>
 <argument name="Users" class="List" usedefault="true">
 <default description="Any user">"*"</default>
 <value>"Homer","Marge","Bart","Lisa","Maggie"</value>
 </argument>
 <argument name="Groups" class="List" usedefault="false">
 <default description="Any group">"*"</default>
 <value>"*\Administrators","Power Users"</value>
 </argument>
 <argument name="Time Period" class="DateTimeRange">
 <value>"0/00/0 00:05:00"</value>
 </argument>
 </arguments>
 ...
```



Notice that the check box next to the Users parameter is cleared, and the default value is used for the parameter. The value itself is not shown, but replaced with a description. The Groups parameter also has a default value, but it is not used in this case.

For details about supported parameter types, see Customizable Parameters.

Pre-Filtering

Use this section for server-side rules, for which this section is executed first on the agent side and then, if matching occurs, on the server side. If an event is matched during pre-filtering, then matching continues in the **<body>** section.

For agent-side rules, this section is ignored.

Enclose the script within the **<prefilter>** tag pair.

Matching

Specify the matching script within the **<body>** tag pair. Your script should do the following:

- Use REL or ECMAScript syntax to describe the situation you are watching out for.
- If the situation occurs, pass on the corresponding event.
- If the situation does not occur, do nothing.

Matching in REL

If your rule uses REL, an expression defines whether the rule is matched, and represents the event record at the same time.

Matching in ECMAScript

If the rule uses ECMAScript, then the script should perform condition testing. In a typical situation, if the result of the test is true, the script should match the event that is passed to it. If the result is false, the script should execute a return statement.

The following three functions are essential for ECMAScript-based rule matching:

1. **OnEvent()**
   This function identifies the incoming event so that the rule can work with it. The argument of the function is the object that represents the incoming event.
2. **match()**
   Call this function when condition testing yields true, and use the same argument as in **OnEvent()**. This ensures that you pass on the same event record that you tested for. This is the preferred use of **match()**, although you can deviate from this practice and modify event fields if necessary.
3. **print()**
   Use this function to debug your scripts. It outputs the values of its arguments to a trace file (**ADCTRACE_RELMatcher***) if the tracing level is set to at least 30.

The trace file is written only if tracing is enabled. The RELMatcher setting in the **adctracer.ini** file controls tracing for this component. For details about tracing configuration, see InTrust Server Tracing.

NOTE: Rules are XML structures, so rule processing involves XML parsing. Script embedding can introduce two types of errors: XML code errors (preventing you from saving the code) and script code errors (causing the rule to fail). Mind that the actual script code to be executed is generated as a result of XML parsing.

When typing or pasting script code inside the <prefilter> or <body> tag pairs, remember that the resulting code must be valid XML. Use metacharacters instead of characters that might break XML markup. On the other hand, make sure that the script code comes out as expected after the parsing.

For more information about XML syntax, see http://www.w3.org/TR/REC-xml/#syntax.

Examples

The following are examples of InTrust rule definitions in ECMAScript and REL.

ECMAScript

```
<rule type="REL" version="1.0" language="jscript">
  <arguments>
    <argument usedefault="false" name="IncludeFiles" description="Specify files that
will be monitored by this rule. Note: you can use path masks." class="List">
      <value>"*"</value>
      <default description="">
         default
      </default>
    </argument>
    <argument usedefault="false" name="ExcludeFiles" description="Specify files that
will be explicitly excluded from monitoring. Note: you can use path masks."
class="List">
      <value>"/etc/syslog.conf"</value>
      <default description="">
        default
      </default>
    </argument>
  </arguments>
  <prefilter>
  </prefilter>
  <body>
function MatchFile( strFileName, arrFileMasks )
{
print( "Matching file name:" + strFileName );
for( i = 0; i &lt; arrFileMasks.length; ++ i )
{
var strMask = arrFileMasks[i];
print( "mask=" + strMask );
strMask = strMask.replace( /\\/g, "\\\\" );
strMask = strMask.replace( /\./g, "\\." );
strMask = strMask.replace( /\?/g, "." );
strMask = strMask.replace( /\*/g, ".*" );
print( "regexp=" + strMask );
var re = new RegExp( strMask, "g" );
if( null != re.exec( strFileName ) )
{
print( "file name matched" );
return true;
}
}
print( "file name is not matched" );
return false;
}
function OnEvent( e )
{
if( "TextFileModified" == e.Action )
{
print( e.Action );
var arrTextFiles = new Array( <parameter name="IncludeFiles"/> );
var arrExceptTextFiles = new Array( <parameter name="ExcludeFiles"/> );
```

```
if( !MatchFile( e.FileName, arrExceptTextFiles )
&amp;&amp; MatchFile( e.FileName, arrTextFiles ) )
{
match( e );
}
}
}
  </body>
</rule>
```

Note the use of XML markup in the definition of the **arrTextFiles** and **arrExceptTextFiles** variables. This is not valid ECMAScript. However after the rule is parsed, the markup is substituted for the values of the relevant rule parameter, producing valid code.

REL

```
<?xml version="1.0"?>
<rule type="REL" version="1.0">
  <arguments>
    <argument name="AdmRightsList" class="List" description="A list of
administrative user rights">
      <value>"SeLoadDriverPrivilege", "SeBackupPrivilege", "SeDebugPrivilege",
"SeRemoteShutdownPrivilege", "SeIncreaseQuotaPrivilege", "SeManageVolumePrivilege",
"SeAuditPrivilege", "SeSecurityPrivilege", "SeTakeOwnershipPrivilege",
"SeTcbPrivilege", "SeSystemtimePrivilege"</value>
    </argument>
    <argument name="Group List" class="List" description="A list of groups for
managing user rights in an organization">
      <value>"Administrators", "Account Operators", "Domain Admins", "Group Policy
Creator Owner", "Enterprise Admins"</value>
    </argument>
  </arguments>
  <prefilter>
(EventID = 608 or EventID = 609)
and striequ( Source, "security" );
  </prefilter>
  <body>
(EventID = 608 or EventID = 609)
and striequ( Source, "security" )
and in( String1, "wi", array(<parameter name="AdmRightsList"/>) )
and not member_of( strcat( String4, "\\", String3 ), array(<parameter name="Group
List"/>), true );
  </body>
</rule>
```

Scripting Response Actions

Response actions are defined for real-time monitoring rules. These actions are performed when rules are matched. InTrust provides the following response action types:

- Execute script
- Execute command
- Run InTrust task
- Send SNMP trap
- Set audit policy

The "Execute script" response action type lets you define and perform your own actions that are not implemented by other response action types.

The script you use can do whatever you need, and it can be in whichever valid language you choose. The scope of the script's actions is defined by the object model available to it.

To configure a custom scripted response action

1. Create a new script object in the **Configuration | Advanced | Scripts** node of the InTrust Manager snap-in.
2. Supply a script that this object must contain.
3. Define and set parameters for the script.
4. Add an "Execute script" response action to the necessary rule, and add the new script object to it.
5. In the response action properties, establish a correspondence between the values that the rule returns and the parameters of the script.

The configuration objects in this brief scenario are explained in the following two subsections.

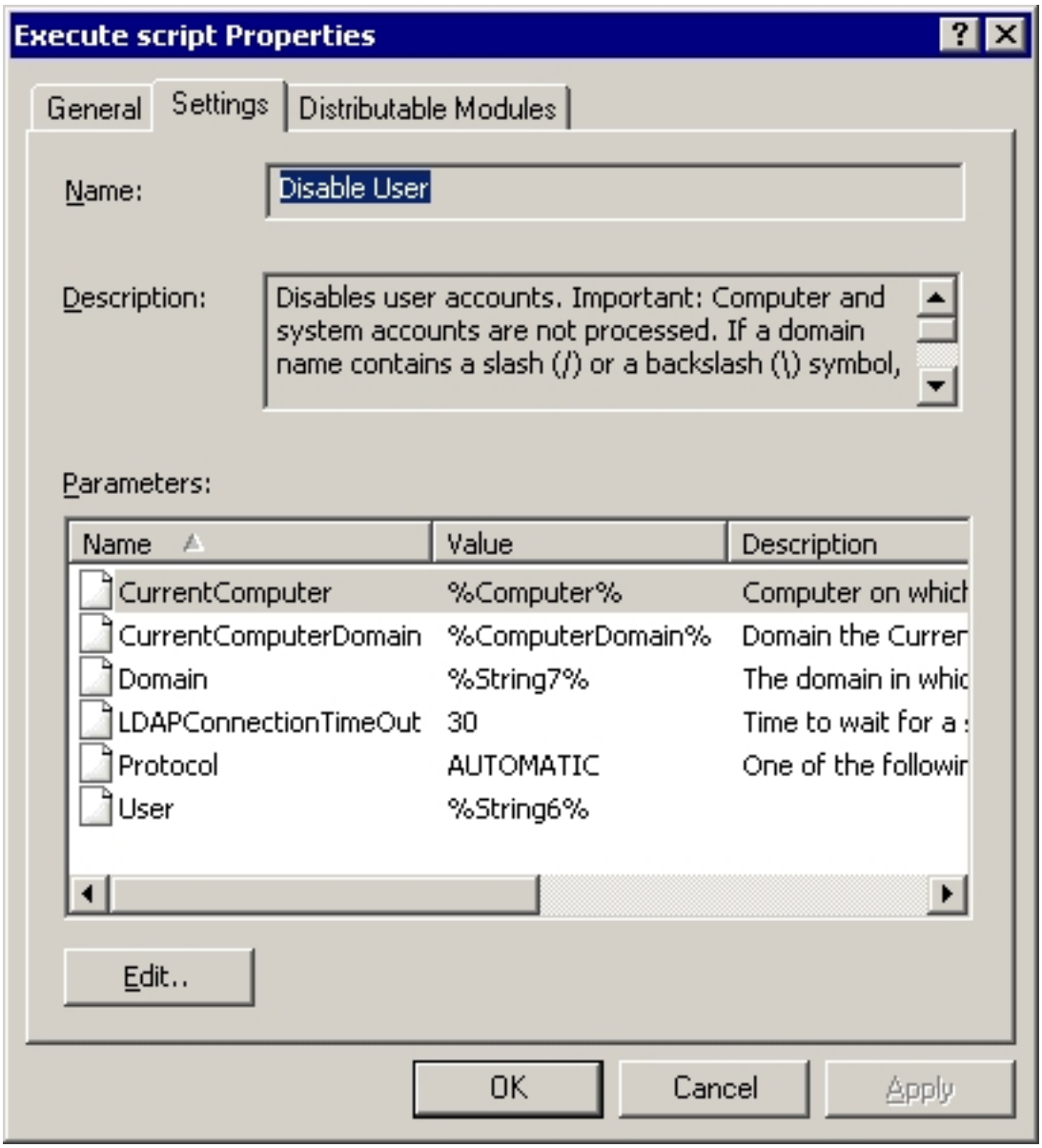
For details about InTrust script objects, see InTrust Script Objects.

"Execute Script" Response Action

Before using your custom script as a response action, you must ensure that the rule passes the correct data to it. The parameter list in the properties of the response action determines what values the rule makes available for use in response action scripts.

To specify parameters that the rule must expose

1. Make sure that the necessary script object exists, and its parameters are configured correctly.
2. Open the properties of the rule you need.
3. On the Response Actions tab of the properties dialog box, click **Add**, and add the script object. When you complete the wizard that starts, the response action properties dialog box opens on the Settings tab.
4. On the Settings tab, use the **Edit** button to specify what values (fixed or dynamic) the rule passes to the script.



Parameters for the list are read from the script object that you selected. At this stage, you assign values to existing script parameters.

Enumerating Sites

You can customize the object enumeration logic for InTrust sites by providing custom enumeration scripts as site objects. To customize site enumeration, create a script object in the **Quest InTrust Manager | Configuration | Advanced | Scripts** container implementing the desired logic, and specify that script for your site.

For details about using script objects, see InTrust Script Objects.

Only ECMAScript is supported for site enumeration scripts. The script needs to implement the **Enumerate (typeItemType)** function. Currently, the value of **typeItemType** can be only **2** (computer).

The function should do the following:

1. Use the **CreateSiteItem(2)** function to create a computer object.
2. When the computer object has been created and returned by **CreateSiteItem()**, set its properties. In a computer object, only the **OriginalName** property needs to be set explicitly. The remaining properties are set automatically.
3. Add the object to the site using the **AddSiteItem(objectSiteItem)** function.
   This function accepts a single parameter: the site item object created on the previous step.
4. To report any possible script runtime errors, use the **ReportError(internalCode, internalMessage, systemCode, systemMessage)** function.

The resulting message is constructed consists of:

- **internalMessage** if **internalCode** equals **E_FAIL**
- **internalMessage + systemMessage** in all other cases

The **systemCode** parameter is reserved.

The message should be visible in the enumeration pane and in session logs if the site enumeration is initiated by InTrust jobs.

Additional features of site enumeration scripts:

- Parameters are supported, as in real-time monitoring response action scripts. The parameter syntax is the same. For details, see the "Execute Script" Response Action section.
- Traces can be output. These traces are written by the **MSNNSiteProvider** component. However, its traces are related not only to site enumeration scripts, so it is a good idea to identify scripted enumeration traces, for example, by using a meaningful prefix in their messages. For details about tracing, see InTrust Server Tracing.
- **#INCLUDE** statements are supported.

The following adds the local host to the site:

```
function AddComputerToSite( strComputerName, SiteObjectType )
{
    #TRACE TraceDebug "enter ->AddComputerToSite"
    var NewSiteItem = CreateSiteItem(SiteObjectType);
    NewSiteItem.OriginalName = strComputerName;
    AddSiteItem(NewSiteItem);
    #TRACE TraceDebug "exit ->AddComputerToSite"
}
function Enumerate( SiteItemType )
{
    AddComputerToSite( "127.0.0.1", 2 );
}
```

In this sample, modify the **Enumerate()** function to do useful work (for example, cycle through computers in an Active Directory domain to find and add those with specific properties).

Language Reference

ECMAScript

ECMAScript is a cross-platform scripting environment that InTrust can use for the following:

- Scripted data sources
- ECMAScript-based rules
- Scripts run by real-time monitoring rules as response actions

When using ECMAScript with InTrust-specific extensions, mind that it does not fully support ActiveX objects.
The ECMA-262 standard defines the ECMAScript scripting language.

JScript

JScript is the implementation of JavaScript by Microsoft, available only on Windows systems. InTrust can use JScript for the following:

- Scripts run by real-time monitoring rules as response actions
- Pre-filtering and matching scripts in rules

REL

REL (Rule Expression Language) is used for defining rules that generate alerts. REL is also used for filtering events.
The result of expression evaluation is usually a Boolean value, or is converted to such a value. This value determines whether a specific alert is generated or message is filtered. The parameters of a REL expression are InTrust events.

Reference

Operators
Functions

Words

Expressions are evaluated left to right. Words are separated by white spaces, tabs and carriage returns. There are six types of words in REL:

- ID
  Name of an argument, field or function. An ID can include alphanumeric characters and the underscore character (_), but cannot begin with a number. IDs are case-sensitive.
- String constant
  Enclosed in single or double quotes. Use the backslash escape character (\) to specify quotes (as in "quoted text: \"some text\"") or the backslash character itself (as in "domain\\user"). String constants, like expressions, use the encoding of the current locale (depending on implementation). String values of fields are converted to the local encoding when they are processed.
- Numeric constant
  Begins with a number or a negative sign (-). If a numeric constant begins with "0x", it is interpreted as a hexadecimal value with characters representing the number bitwise (including the sign bit). All numeric constants are signed 32-bit integers.
- Boolean constant
  True or false.
- Operator
  The following operators are used:
  - + - * / % > >= < <=
  - = != () | & << >> ? :
  - . [] or and not
- Auxiliary character

Characters that are part of expression language syntax, such as parentheses, curly braces, and the function definition character.

Formal Language Grammar

```
expression ::= +(token)

token ::= [*space] (identifier | string | number | boolean | operator |
special) [*space]

space ::= " " | "\t" | "\n"

identifier ::= (alpha | "_") *(alpha | numeric | "_")

string ::= single_quoted | double_quoted

single_quoted ::= "'" *(char | 'r;"' | "\'" | "\\") "'"

double_quoted ::= 'r;"' *(char | "'" | 'r;\"' | "\\") 'r;"'
```

```
char ::= 0x00 | …… | 0xFF ; all characters except 'r; " \

number ::= (["-"] +numeric) | ("0x" +hex)

boolean ::= "true" | "false"

operator ::= plus | minus | multiply | divide | modulus | greater | greater_equal
| less | less_equal | equal | inequal | bit_or | bit_and | bit_xor | shift_left
| shift_right | not | or | and | conditional | selection | field_selection |
open_sq | close_sq

plus ::= "+"

minus ::= "-"

multiply ::= "*"

divide ::= "/"

modulus ::= "%"

greater ::= ">"

greater_equal ::= ">="

less ::= "<"

less_equal ::= "<="

equal ::= "="

inequal ::= "!="

bit_or ::= "|"

bit_and ::= "&"

bit_xor ::= "^"

shift_left ::= "<<"

shift_right ::= ">>"

not ::= "not"

or ::= "or"

and ::= "and"

conditional ::= "?"

selection ::= ":"

field_selection ::= "."

open_sq ::= "["

close_sq ::= "]"

symbol ::= open_br | close_br | comma | definition | open_cr | close_cr

open_br ::= "("

close_br ::= ")"

comma ::= ","
```

```
definition ::= ":="

open_cr ::= "{"

close_cr ::= "}"

Formal Language Syntax

REL_expression ::= *(function_definition) expression

function_definition ::= "def" identifier open_br [ argument_def_list ] close_br
definition open_cr expression close_cr

argument_def_list ::= argument_def *(comma argument_def)

argument_def ::= identifier [ equal expression ]

primary_expression ::= constant | identifier | function |

(open_br expression close_br)

constant ::= string | number | boolean

function ::= identifier open_br [argument_list] close_br

argument_list ::= expression *(comma expression)

operator_indexing ::= primary_expression |

operator_indexing field_selection identifier |

operator_indexing open_sq expression close_sq

operator_unary ::= primary_expression |

(minus | not) primary_expression

operator_multiply ::= operator_unary |

operator_multiply (multiply | divide | modulus) operator_unary_minus

operator_add ::= operator_multiply |

operator_add (plus | minus) operator_multiply

operator_shift ::= operator_add |

operator_shift (shift_left | shift_right)  operator_add

operator_bit_and ::= operator_shift |

operator_bit_and bit_and operator_shift

operator_bit_xor ::= operator_bit_and |

operator_bit_xor bit_xor operator_bit_and

operator_bit_or ::= operator_bit_xor |

operator_bit_or bit_or operator_bit_xor

operator_compare ::= operator_bit_or |

operator_compare (greater | greater_equal | less | less_equal) operator_bit_or

operator_equal ::= operator_compare |
```

```
operator_equal (equal | inequal) operator_compare

operator_and ::= operator_equal | operator_and and operator_equal

operator_or ::= operator_and | operator_or or operator_and

operator_conditional ::= operator_or |

operator_or conditional expression selection expression

expression ::= operator_conditional
```

In REL-based event filters, there must be no semicolon at the end of the expression. In rules, the semicolon is required.

Expressions

An expression is a basic language entity. An expression has a value and a type. It consists of constants, events and event fields, and also operators and function calls required for calculations.

Expression examples:

```
5
A+B
Source
Count<5
strstr(Description, "found")
```

Expression Types

Simple types

- string
- number
- boolean
- empty

Complex types

- message (event)
- array

Arrays

Use brackets to specify the array element you want to access. The first element of an array has an index of 0.

Arrays can include other arrays. For example, it is possible to return an array of field arrays to denote a set of fields each of which has a set of auxiliary fields.

If you go outside the scope of an array, the returned value is empty (an empty field or a message without fields). Arrays can also be empty; in this case, getting any element of the array returns an empty value.

Type Conversion

REL does not have strong typing. All expression values are converted to other types as necessary, according to the rules described further on.

- Strings to numbers
  Characters are interpreted as a number. If the expression begins with "0x", the remaining characters are interpreted as a hexadecimal representation of a number. If the string does not represent a number, zero is returned.
- Numbers to strings
  A number is converted to its decimal representation without leading spaces or zeros.
- Booleans to strings and numbers
  Boolean values are converted to the strings "true" and "false", and to the numbers 1 and 0, respectively.
- Numbers to Booleans
  False if the number is 0, and true otherwise.
- Strings to Booleans
  False if the string is empty.

Converting Arrays

Wherever an array is required, a simple value can substitute it, and the reverse is also true. Conversions are made according to the following rules:

- Simple value to array
  A message is converted to a message array with a single element. A field is converted to a field array with a single element.
- Array to simple value
  The first element of the array is returned. In multi-dimensional arrays, this rule is applied recursively multiple times and returns the element which is the first at all nesting levels.

Empties

Empty is a special type used to represent undefined values.

Conversion of empties

- Empty to number
  0 is returned.
- Empty to string
  Empty string (same as "").
- Empty to Boolean
  False.
- Empty to array
  Array with one element (empty).

Events and Fields

An event is an implicit parameter in a REL expression. Event fields are accessed by name. Example:

```
EventID = 100
```

If an event or event array is returned by a function, you can access its fields using the "." operator. Example:

```
previous_lim(Z.EventID=100, "1:00:00").String1
select(Z.EventID=100, "1:00:00")[0].String1
```

If the specified field is not found, a value of the "empty" type is returned.

The "." operator can be applied to an event array. In this case, the result is an array of values of the specified field. The size of this array is equal to the size of the event array. Example:

```
select(Z.EventID=100, "1:00:00")._LocalTime
```

An event cannot be converted to a simple (scalar) type or an array. This operation returns an empty.

The asterisk character (*) is a special case of field name. This field contains an array of the string values of all message fields.

Operators

The following table shows the operators that REL supports. If an expression is an operator argument and the expression value type is invalid, the value is converted to the necessary type according to standard rules (See Type Conversion).

SYNTAX	OPERAND TYPE	VALUE TYPE	RESULT
Access and indexing operators			
expr1 [expr2]	Number array	Field or message	Provides access to an element of a field array or message array.
expr1 . id	Array, ID	Field array	Provides access to message fields or auxiliary fields of a particular field. The **expr1** part can be a single message or a field (See Type Conversion).
Math operators			
- expr1	Number	Number	A negative value of **expr1** defined as (0 – expr1).
expr1 + expr2	Number	Number	Sum of operands.
expr1 – expr2	Number	Number	Difference of operands.
expr1 * expr2	Number	Number	Product of operands.
expr1 / expr2	Number	Number	Ratio of **expr1** and **expr2**. In case of overflow (and division by zero) the result is the maximum possible integer (2^31-1 or –2^31).
expr1 % expr2	Number	Number	Excess of integer division defined as **(expr1 – expr1 / expr2 * expr2)**.
expr1 & expr2	Number	Number	Result of a bitwise AND operation on operands.
expr1 | expr2	Number	Number	Result of a bitwise OR operation on operands.

SYNTAX	OPERAND TYPE	VALUE TYPE	RESULT
expr1 ^ expr2	Number	Number	Result of a bitwise XOR operation on operands.
expr1 << expr2	Number	Number	Result of a bitwise signed left shift of the value of **expr1** by **expr2** bits.
expr1 >> expr2	Number	Number	Result of a bitwise signed left shift of the value of **expr1** by **expr2** bits.
Comparison operators			
expr1 = expr2, expr1 != expr2	Simple	Boolean	True if operand values are equal or not equal, respectively; false otherwise. If operand types mismatch, the second operand's type is converted.
expr1 < expr2, expr1 <= expr2, expr1 > expr2, expr1 >= expr2	Simple	Boolean	True if the first operand is less, less than or equal to, greater, or greater than or equal to the second, respectively; false otherwise. If operand types mismatch, the second operand's type is converted. String comparison is case-sensitive. In Boolean value comparisons, false is always less than true.
Boolean operators			
not expr1	Boolean	Boolean	Returns the negation of **expr1**.
expr1 and expr2	Boolean	Boolean	True if both expressions yield true; false otherwise. The value of **expr1** is calculated first. If this value is false, **expr2** is not calculated.
expr1 or expr2	Boolean	Boolean	True if at least one of the expressions yields true. The value of **expr1** is calculated first. If this value is true, **expr2** is not calculated.
expr1 ? expr2 : expr3	expr1: Boolean expr2: any type	Boolean	The value of the expression is **expr2** if **expr1** is true; otherwise, the value is **expr3**. Either way, only one expression is calculated. This operator accepts all types, including messages, fields and arrays. The expressions **expr2** and **expr3** can be of different types. In this case, the resulting type depends on the result of **expr1**.

Operator Precedence

The order of operator processing can be changed by enclosing expressions in parentheses. The access and indexing operators have the highest precedence, math operators come next:

```
[] .
- (unary) not
* / %
+ -
<< >>
&
```

```
^
|
```

Comparison operators follow:

```
< > <= >=
= !=
```

Boolean operators have the lowest precedence:

```
and
or
?:
```

Operator usage examples:

- 18+44 (result: 62)
- "14"/3 (result: 4)
- true=2 (result: true)
- 2=true (result: false)
- Z._LocalTime (result: local time of event Z)
- ""?14:false (result: false)
- "aaa"<"bb" (result: true)
- (3>=2)+1 (result: 2, number(3>=2)=1)

Functions

A function call is a function ID followed by a list of arguments in parentheses, which can be empty. The value of such an expression is the value returned by the function. Functions can be built-in or custom.

The built-in function library includes string operations, math operations, access to the event store and so on. A detailed list of built-in functions is given further on.

Custom Functions

The syntax for defining custom functions is as follows:

```
def name([argument_list]) := { expression }
```

The general ID naming rules apply to names of internal functions. The list of internal function arguments (if any) is a comma-separated list of parameter names with optional default values. The arguments of internal functions can be constants, fields and field arrays, and messages. The default value is separated from the argument name by the equals character (=) and is assigned to the argument automatically if the argument is omitted in a function call. You can omit any number of arguments starting from the right, as long as the arguments have default values. The expression in curly braces is the result of the function call.

Example:

```
def DoDivide(a, b=10) :=
{
a/b
}
def IsLogon() :=
```

```
{
in_range( EventID, "529-537, 539, 548-549" )
and striequ( Source, "security" )
}
```

Custom and built-in functions share the same namespace. Redefining a function results in an error.

When functions are called, whether built-in or custom, the types of the arguments are not checked. When types mismatch, standard type conversion is performed.

Built-In Function Library

REL's built-in function library includes the following function types:

- General Functions
- Math Functions
- String Processing Functions
- Date and Time Processing Functions
- Message Selection and Filtering Functions
- Function for Defining Missing Messages
- Windows-Specific Functions

Function syntax can be described by the following template:

```
type_of_returned_value function_ID (argument1_type argument1, argument2_type
argument2, ...)
```

where types are specified as string, number, Boolean or message. Brackets ([]) after the argument type specify that the function treats the argument as an array and processes all of its elements. If you specify **any-type** as the argument type, the function can take arguments of any type.

If the function takes an indefinite number of arguments, this is specified as follows: "**type name1, type name2, ...**", where type can be any word that describes the meaning of the arguments.

Built-in functions do not check argument types; they simply treat arguments as expressions of the specified type.

General Functions

SYNTAX	DESCRIPTION
number count(any-type[] expr1)	Returns the number of array elements. For simple types, the returned value is always **1**.
boolean empty(any-type[] expr1)	Returns **true** if array **expr1** is empty. For simple types, **false** is always returned. Defined as **count(expr1)=0**.
boolean exist(any-type[] expr1)	Returns **true** if array **expr1** has elements. For simple types, **true** is returned. Defined as **not empty(expr1)**.
number number (any-type expr1)	Converts the expression to the **number** type. Type conversion functions perform standard conversion and are mainly used for explicitly setting argument types in such functions as **equal()**, **differ()** or **max()**.

SYNTAX	DESCRIPTION
string string(any-type expr1)	Converts the expression to the **string** type.
boolean boolean (any-type expr1)	Converts the expression to the **Boolean** type.
boolean equal(any-type[] expr1)	Returns **true** if all array elements are equal.
boolean differ(any-type[] expr1)	Returns **true** if all array elements differ. That is, there are no equal value pairs. Note that the negative result of an **equal()** is required but not sufficient for the positive result of a **differ()** with the same arguments.
min(number[] expr1)	Returns the least of the array items.
max(number[] expr1)	Likewise, returns the greatest of the array items.
boolean in_range (any-type exp, string range)	Checks whether **exp** is within **range**; **range** is a string of values which are either comma-separated or specified as ranges. The values can be only positive integers. Example: `in_range (EventID, "451, 512-654").`
boolean in (string exp, string options, string[] values)	Tests whether **exp** is listed in values, which includes strings (literals or regular expressions). The **options** string specifies how string matching is performed: • **[b|e|c]** **b**—basic regex **e**—extended regex **c**—simple comparison (**strcmp**); this is the default • **[i|s]** **i** —case-insensitive **s**—case-sensitive (default) • **[w]** use wildcards Example: `in( strcat( String4, "\\", String3 ), "wi", array ("DOMAIN1\\guest", "DOMAIN2\\*") )`

Math Functions

SYNTAX	DESCRIPTION
number abs(number expr)	Returns the absolute value of the argument.
number sqr(number expr)	Returns the square root of the numeric value of **expr**.

SYNTAX	DESCRIPTION
number sqrt(number expr)	Returns the square root of the numeric value of **expr**. If the value of **expr** is negative, **–1** is returned.
number sum(number[] expr1)	Returns the sum of all arguments. If the argument is an array, its value is the sum of all its elements.
number product(number[] expr1)	Returns the product of all arguments.
number difference (number[] expr1)	Returns the numeric difference between the greatest and least elements in the argument sequence. Defined as **(max(expr1)-min(expr1))**.
number average(number [] expr1)	Returns the arithmetical mean of arguments. This is the equivalent of **sum(expr1)** divided by the number of arguments, including array elements.
number deviation(number pos, number[] expr1)	Returns the statistic deviation of the **pos**-th selection element (element indexing starts with 0). Defined as **expr_at_pos – average(expr1))**.
number mean_deviation (number[] expr1)	Returns the mean statistic deviation of the selection. Defined as **average(abs (deviation(1, expr1)), abs(deviation(2, expr1)), ...)**.
number dispersion (number[] expr1)	Returns the statistic dispersion of the selection. Defined as **average(sqr(expr1 [0] – average(expr1), sqr(expr1[1] – average(expr1), ...)**.
number std_deviation (number[] expr1)	Returns the mean square deviation of the selection. Defined as **sqrt(dispersion (expr1))**.

String Processing Functions

SYNTAX	DESCRIPTION
number strlen (string expr1)	Returns the number of characters in the string **expr1**.
string strcat (string expr1, string expr2, ...)	Returns the string value which is a concatenation of all specified strings.
string substr (string expr1, number pos, number length)	Returns a substring of the string **expr1**, beginning with the **pos**-th position and having a length of length. Characters are numbered starting with 0. If length is –1, everything from **pos** to the end of the string is returned.
number strstr (string expr1, string expr2)	Returns the position of the first encounter of **expr2** in **expr1**. If **expr1** does not contain **expr2**, **–1** is returned.
string strupr (string expr1)	Returns the string **expr1** where all characters are converted to upper case according to the rules of the local encoding.
string strlwr	Returns the string **expr1** where all characters are converted to lower case according to the

SYNTAX	DESCRIPTION
(string expr1)	rules of the local encoding.
number stricmp (string expr1, string expr2)	Compares two strings while ignoring case. Returns the following numeric values: **<0** if **expr1** < **expr2** **0** if **expr1** = **expr2** **>0** if **expr1** > **expr2**
boolean striequ(string expr1, string expr2)	Returns **true** if two strings are identical. Ignores case. Equivalent to **stricmp(expr1, expr2) = 0**.
number[][] regexp(string exp, string str, string options)	Runs regular expression **exp** on string **str**. Supported regular expressions are defined by IEEE Std 1003.1-2001 (so-called POSIX regular expressions). Both basic and extended regular expressions are supported. The text value of options can contain the following characters: • **e**—specifies that **exp** is an extended regular expression (ERE) • **b**—specifies that **exp** is a basic regular expression (BRE) • **n**—boost::regbase::normal (JavaScript) – default • **i**—ignore case The function returns a two-dimensional Nx2-sized array containing all substrings matching the regular expression, where N is the total number of matches. The first element of each nested array contains the index of the string start (indexing starts with 0); the second element contains the string length.

Date and Time Processing Functions

The event creation date and time (Time field) are represented by a string according to the description of the field. Date and time processing functions convert this string to numbers and back, and do math operations on the date and time. If it is specified that the function returns time, it actually returns a string formatted for the Time field.

SYNTAX	DESCRIPTION
number year (string expr)	Returns the numeric year value.
number month (string expr)	Returns the numeric month value.
number day (string expr)	Returns the numeric day value.
number hour (string expr)	Returns the numeric hour value.
number minute (string expr)	Returns the numeric minute value.

SYNTAX	DESCRIPTION
number second (string expr)	Returns the numeric second value.
number ms (string expr)	Returns the numeric millisecond value.
number dow (string expr)	Returns the numeric day-of-the-week value (**0** is Sunday, **1** is Monday, and so on).
string local_time ()	Returns the current system time in local time format.
string time_to_ local(string expr)	Converts GMT time to local time format of the current system.
string time_to_ gmt(string expr)	Does a conversion that is the reverse of **time_to_local()**.
string time (number month, number day, number year, number hour, number minute, number second, number ms)	Returns a string representation based on numeric month, day, year, hour, minute, second and millisecond values of event creation time. You can omit any number of fields. In this case, general rules apply, and the values of omitted fields are assumed to be 0.
string time_diff (string expr1, string expr2)	Returns the string representation of the date and time based on the date difference between **expr1** (later event) and **expr2** (earlier event). Afterwards, you can use the functions **year()**, **month()** and **day()** to find out how many units of time elapsed between the events. Example: `time_diff("4/28/2002", "3/25/2001")="1/3/1"` (one year, one month, and three days)
number [xxx]s_ diff(string expr1, string expr2)	This family of functions returns the difference between two events (**expr1** is the later event) in the number of years, months, hours, minutes, seconds and milliseconds. Afterwards, you can use the resulting values to construct non-normalized time (see **time_add()**).
string time_add (string expr1, string expr2)	Adds the time **expr2** to the time **expr1**; **expr2** can be non-normalized. Non-normalized time means such time representation in which the values of particular fields exceed threshold values (for example, 44 hours and 120 minutes). Normalization means conversion to conventional representation (for the previous example, 1 day and 22 hours). Normalization in general is impossible without a point of reference, because it is not known how many days there are in a month or whether a year is a leap year. Therefore, it applies only to the **time_add()** function. Examples of date and time processing functions: `strcat(day(_LocalTime), ".", month(_LocalTime))` Represents the time of event in "day.month" format.

SYNTAX	DESCRIPTION

	`minutes_diff(_LocalTime, _GMT)`
	Time zone in minutes.
	`time_to_local(time_add(_GMT, time(0, 0, 0, 1, 30)))`
	`time_to_local(time_add(_GMT, time("1:30")))`
	`time_to_local(time_add(_GMT, time("0:90")))`
	Represents one and a half hours since the event was generated in the server's local time format.

Message Selection and Filtering Functions

These functions provide access to a store of received real-time monitoring messages and perform array filtering. The two function categories interoperate closely, because conditions for selection and filtering are somewhat different, and these functions complement each other.

Message selection functions do not consider the message that is currently being processed.

Mind that these functions evaluate each tested expression within its own scope rather than in the scope of the current expression. Such expressions are evaluated for each message that comes in from the data source, and must return true if the message meets selection conditions. The tested condition is the **Z** parameter of the expression, and its fields are accessed using the "Z." prefix.

Expressions used in selection functions do not have access to the parameters of the root expression. In these expressions, you cannot use message selection and filtering functions. In other words, such expressions can only set conditions, which are independent of the scope of the expression currently being processed.

The reason for such limitations is selection optimization. Events that pass the filter are stored in a buffer and returned when functions are called.

No such limitations exist for expressions that are parameters of the filtering function. Access to the processed message is performed the standard way, while the filtering element is stored in the **Z** parameter. Thus, complex selection rules are implemented in two stages:

1. Messages are selected by independent parameters.
2. Filtering rules of any complexity are applied.

SYNTAX	DESCRIPTION
message[] select(expr1, string time)	Selects all messages that meet condition **expr1** and are within a time interval no less than time relative to the current message. The time string is formatted for the **Time** field (see Date and Time Processing Functions). If no matching message is found, an empty array is returned. The condition **expr1** is evaluated before the main expression. Thus, if the current event meets this condition, it will be present in the array of events returned by the **select()** function. The events in the array are sorted in ascending order by arrival time, so **select()[0]** is the first (oldest) event. When the rule is matched, the **select()** function's event buffer is freed. If the buffer was not empty when the match occurred, then the filter will be applied to subsequent events and stored in the buffer starting with the time expressed as **OldestEvent._LocalTime + period**, where **period** is the time interval set in **select()**.

SYNTAX	DESCRIPTION
	This prevents the rule from being matched too frequently in case there is an intense flow of events that meet the conditions of the **select()** function.
message[] previous(expr)	Returns the previous message that meets the specified condition. The function's parameters are identical to the parameters of **select()**. The difference is that this function does not consider event generation time. The limitations for **expr1** are the same as in **select()**. If no matching message is found, an empty array is returned. The condition **expr1** is evaluated after the main expression. This means that the current event can never be the result of this function. If the rule is matched, the event stays in the buffer.
message[] previous_ lim(expr1, string time)	This version of the **previous()** function returns the message that meets the specified condition and occurred after the specified time (**time**). Condition evaluation occurs in the same way as for **previous()**.
message[] consolidate (expr1, string id1, string id2, ...)	Consolidates message array **expr1** based on the fields **id1**, **id2**, ... This means that any two or more messages with identical values for the specified fields are merged together. For the other fields, values are taken from the latest message. **any-type[] filter(any-type[] expr1, expr2)** evaluates **expr2** for each element of the one-dimensional **expr1** array in a row. Returns an array of elements for which the condition is met. The **Z** parameter is used to refer to an array element.
message[] select_ filtered(expr1, expr2, string period)	This function is almost identical in effect to the following construction: **filter(select(expr1, string period), expr2)**. The main difference is that when the rule is matched, only those events are extracted that pass both the **expr1** and the **expr2** filter. **expr1** is a filter with a condition that does not depend on the current event (generalized filter), and **expr2** filters the current event (context filter). **expr2** uses temporary variable **Z**, which iterates through each array element (as in the **filter()** function) Example: `count(select_filtered(EventID=100, Z.User = User, 10:00)) >= 10` This stores all events where EventID is 100. As soon as there are 10 events about a particular user, the rule is matched, and only events about this user are removed from the buffer. In contrast, using **select()** has the disadvantage of removing all events.

message[] select_ matches(expr1, string period)	When the rule is matched, the event array is placed in a buffer with expiration time set to **period**.
	This function is mainly intended to prevent multiple rule matches.
	Example:
	`count(select_filtered(EventID=100, Z.User = User, 10:00)) >= 10;`
	With this expression, if there are 100 events about the same user, the rule is matched 10 times. To limit the number of matches, rewrite it as follows:
	`count(select_filtered(EventID=100, Z.User = User, 10:00)) >= 10 and empty (select_matches(Z[0].User = User, 10:00));`

SYNTAX	DESCRIPTION

	This ensures the rule is matched no more frequently than once in 10 minutes.
	Examples of message database and filtering function usage:
	`count( filter( select( Z.EventID = 100, "1:00:00" ), Z.User = User ) ) > 5;`
	The condition is met if there have been more than five events during the past hour where the EventID is 100 and the **User** field has the same value as in the current event.
	`EventID=100 and not exist(previous_lim(Z.EventID=200,"00:10:00"));`
	The condition is met if an event with EventID=100 has come in and there have been no events with EventID=200 in the past 10 minutes.

Function for Defining Missing Messages

This function defines real-time messages that were not received when expected. The function can consider absolute time and relative time intervals.

SYNTAX	DESCRIPTION
boolean missing(expr condition, string start_ time, string duration)	This function returns **true** if the event that matches **expr** did not occur during one of the specified time intervals. Arguments: 1. **condition**—testing condition 2. **start_time**—specified in the cron format as five numbers separated by commas or tabs: • minute (0-59), • hour (0-23), • day of the month (1-31), • month of the year (1-12), • day of the week (0-6 with 0=Sunday). 3. **duration**—duration of the time interval; the format is the same as for **select()** and **previous_lim()** Example: `missing(Z.Source="Backup", "0 1 * * 4,6", "1:00");` Specifies that an event from the backup system is expected every Wednesday and Saturday between 1:00 and 2:00 AM.

Windows-Specific Functions

SYNTAX	DESCRIPTION
boolean direct_ member_of(string user, string groups [], [boolean any_ group], [boolean default])	This function returns **true** if the user **user** is a direct member of one of the groups listed in the **groups** array. The **any_group** parameter specifies whether membership in one of the groups (true) or all of the groups (false) is verified. The **user** and **groups** strings can be names or SIDs (which start with "S-1.") The **default** parameter is the value that the function returns if membership verification fails (for example, if access is denied.) Either or both of the last two parameters can be omitted. The following default values are assumed: **any_group** = false, **default** = false

boolean is_ current_user(string domain, string username)	Returns **true** if **domain** and **username** are the same as the account that is calling the function. If **domain** is an empty string, then **username** can be the SID of the account you want to check.
	`boolean is_current_logon_session(string session)`
	Returns **true** if the specified string matches the current logon session. The string must have the following format: **(<hex_num>, <hex_num>)**
	Example:
	`(0x0,0x3E7) (as in EventLog)`

boolean in_OU(string domain, string name, string[] orgUnits, [boolean anyGroup], [boolean defaultValue])	Tests whether the specified account (**domain**, **name**) belongs to the specified Active Directory organizational units (specified as plain OU names, such as "Accounting" and "Human Resources"). If **domain** is specified, then **name** is the account name. Otherwise, **name** can be the principal name or SID. The **anyGroup** and **defaultValue** parameters are used as in the **direct_member_of()** function. By default, **anyGroup** = true, **defaultValue** = false
boolean member_of(string user, string [] groups, [boolean any_ group], [boolean default])	This function returns **true** if user is a direct or indirect member of the specified groups. The **any_group** and **default** parameters are used as in the **direct_member_of()** function. By default, **anyGroup** = false, **defaultValue** = false.
number get_ account_type(string computer, string account)	Queries the type of the specified account (**account**) on the specified computer (**computer**). If the value of **computer** is an empty string, the local computer is assumed. The account type is determined through **NetUserGetInfo**. • UF_TEMP_DUPLICATE_ACCOUNT 0x0100 • UF_NORMAL_ACCOUNT 0x0200 • UF_INTERDOMAIN_TRUST_ACCOUNTv 0x0800 • UF_WORKSTATION_TRUST_ACCOUNT 0x1000 • UF_SERVER_TRUST_ACCOUNT 0x2000

SYNTAX	DESCRIPTION
	Additional flags: • UF_SCRIPT 0x0001 • UF_ACCOUNTDISABLE 0x0002 • UF_HOMEDIR_REQUIRED 0x0008 • UF_LOCKOUT 0x0010 • UF_PASSWD_NOTREQD 0x0020 • UF_PASSWD_CANT_CHANGE 0x0040 • UF_ENCRYPTED_TEXT_PASSWORD_ALLOWED 0x0080 If the type query fails, **0** is returned.

boolean is_primary_group(string user, string [] groups)	Tests whether one of the specified groups is the primary group of the specified user (**user**).
	Examples:
	`(EventID = 632 or EventID = 636)`
	`and striequ( Source, "security" )`
	`and not is_current_user( String7, String6 )`
	`and in( strcat( String4, "\\", String3 ), "wi", array("*\\Guests", "*\\Domain Guests") );`
	The expression tests whether the account used for adding a user to the "guests" group is different from the InTrust Server account.
	`(get_account_type(Computer, Account) &`
	`(UF_WORKSTATION_TRUST_ACCOUNT | UF_WORKSTATION_TRUST_ACCOUNT)) = 0`
	Tests whether the account is a user not a computer.

Object Library

InTrust functionality extensions rely on the following:

- Conventional scripting host object models
  For example, on Windows, all Windows Script Host objects are available to ECMAScript-based rules.
- Additional InTrust-specific objects and functions for auditing and real-time monitoring

Reference

For details, see the following topics:

- Standard Objects
- Emulated Standard Objects
- InTrust-Specific Objects

Standard Objects

On Windows computers, Windows Script Host objects are available to InTrust (where applicable). For more information, refer to MSDN resources.

On other platforms, it is recommended that InTrust have access to the ECMAScript object model. The following ECMAScript objects are especially useful for InTrust operations:

- FileSystemObject
- ShellObject
- SystemInformationObject
- RegKey
- ActiveXObject
- Enumerator
- VBArray

For details about these objects, refer to MSDN or the ECMA-262 specification, depending on the object model you are using.

Emulated Standard Objects

For custom scripted data sources and custom text log data sources, InTrust provides a number of special objects. These objects are look-alikes of standard WSH objects and can be accessed by the scripts that the data sources are based on.

Note that these objects can be used only for InTrust gathering and real-time monitoring purposes. They do not exist outside the scope of InTrust operations.

- FileSystemObject
- Folder
- File
- TextStream
- ShellObject
- ScriptExecObject
- RegKey
- SystemInformationObject
- TimeOfDayInfoObject
- ActiveXObject

Example

The following script displays a list of files and subfolders in the **C:\Temp** folder. Then it opens a text file and types it line by line.

Note that **print()** is a sfunction from the rule object model. It is not available outside the scope of rules.

```
try
{
    var FS = new FileSystemObject();
    var folders = FS.GetFolder("c:\\temp\\").SubFolders;
    print("> dir c:\\temp\n");
    for (var i=0; i<folders.length; ++i){
        print("["+folders[i].Name+"]\n");
    }
    var files = FS.GetFolder("c:\\temp").Files;
    for (var i=0; i<files.length; ++i){
        print(files[i].Name+"\n");
    }
    print("---------\n");
    var f = FS.OpenTextFile("example.txt",
                FileSystemObject.AccessMode.ForReading,
                FileSystemObject.CreationDisposition.OpenExisting, "Auto");
    var i = 1;
    while (! f.AtEndOfStream ) {
        s = f.ReadLine();
        print(i + "] " + s + "\n");
        ++i;
    }
}
catch (err)
{
    print("Error: " + err + "\n");
}
```

ActiveXObject

This class lets you use COM automation in scripts. It has the same functionality as the comparable class in Windows Script Host.

You can create ActiveX objects as follows:

```
// Method 1:
var p = new ActiveXObject("{CB3DF937-8DBB-4130-8A11-D47E12F61315}");
// Here, the argument is the CLSID
// of the object you want to create.
//Method 2:
var p = new ActiveXObject("Scripting.FileSystemObject");
// ProgID is the argument.
```

After you have created the object, you can access its methods, read and write its properties.

In case of error, an exception is generated, represented by an **ActiveXError** object. This object class contains the following fields:

- **number**
  Error code
- **text**
  Message that corresponds to the error code
- **description**
  An optional description of the error

In the following example, possible exceptions are caught:

```
try
{
var p = new ActiveXObject("SomeCOMObject.ProgID");
p.method();
}
catch( err )
{
      print( "Error: " + err );
}
```

File

NAME	TYPE	DESCRIPTION
OpenAsTextStream([iomode[, crmode[, format]]])	method	Returns a new **TextStream** object. See **FileSystemObject.OpenTextFile** for details.
Delete()	method	Deletes the file.
Name	property	Returns the name of the file.
Path	property	Returns the full path to the file.
DateLastModified	property	Returns the date and time that the file or folder was last

NAME	TYPE	DESCRIPTION
		modified.
Size	property	Returns the size of the file in bytes.

FileSystemObject

NAME	TYPE	DESCRIPTION
OpenTextFile (absolute_path[, iomode[, crmode[, format]]])	method	Opens the specified file and returns a TextStream object that can be used to read from the file. Requires the absolute path to the file as a parameter. • The **iomode** parameter can be one of the constants returned by the properties of the object, which are accessed by the **AccessMode** static property. • The **crmode** parameter can be one of the constants returned by the properties of the object, which are accessed by the **CreationDisposition** static property. • The **format** parameter specifies the encoding of the file. This can be one of the following string values: "MBCS", "UTF-8", "UTF-16", "UTF-16LE", "UTF-16BE", "Auto". See the next table for details.
GetFile(path)	method	Returns a **File** object representing the file with the specified path.
GetFolder(path)	method	Returns a **Folder** object representing the folder with the specified path.
DeleteFile(path)	method	Deletes the specified file.
CreationDisposition	static property	Returns an object with the **OpenExisting** read-only property.
AccessMode	static property	Returns an object with the **ForReading** read-only property.

Use "MBCS", "UTF-8", "UTF-16LE" or "UTF-16BE" only if you know for certain what file encoding is used. Otherwise, use "UTF-16" (platform specific) or "Auto". This is the default, which uses the byte order mask (BOM) to detect file encoding.

The following table shows how the encoding is determined from the BOM and explicitly specified format parameter.

	"MBCS"	"UTF-8"	"UTF-16"	"UTF-16LE"	"UTF-16BE"	"AUTO"
<NONE>	MBCS	UTF-8	UTF-16 p. s.	UTF-16LE	UTF-16BE	MBCS
UTF-8	MBCS	UTF-8	UTF-16 p. s.	UTF-16LE	UTF-16BE	UTF-8
UTF-16LE	MBCS	UTF-8	UTF-16LE	UTF-16LE	Error	UTF-16LE
UTF-16BE	MBCS	UTF-8	UTF-16BE	Error	UTF-16BE	UTF-16BE

*In the table "p. s." stands for platform-specific byte order.

Folder

NAME	TYPE	DESCRIPTION
Files	array of **File** objects	Provides a list of files contained within the folder.
SubFolders	array of **Folder** objects	Provides a list of folders contained within a folder.
Name	property	Returns the name of the folder.
Path	property	Returns the full path to the folder.

RegKey

NAME	TYPE	DESCRIPTION
RegKey([key])	constructor	Constructs a new **RegKey** instance.
Connect(key, computer)	method	Call this method to connect to the key of the remote registry.
Open(key, subkey)	method	Call this method to open the specified subkey.
GetDWORD(valuename)	method	Retrieves a DWORD value.
GetString(valuename)	method	Retrieves a string value.
GetMultiString(valuename)	method	Retrieves a multiline string value and returns an array of the strings.
IsValid	property	Indicates whether the registry key is valid.
HKEY_CLASSES_ROOT	static property	Returns a new **RegKey** object pointing to HKCR key.
HKEY_CURRENT_USER	static property	Returns a new **RegKey** object pointing to HKCU key.
HKEY_LOCAL_MACHINE	static property	Returns a new **RegKey** object pointing to HKLM key.
HKEY_USERS	static property	Returns a new **RegKey** object pointing to HKU key.
HKEY_PERFORMANCE_DATA	static property	Returns a new **RegKey** object pointing to HKPD key.
HKEY_CURRENT_CONFIG	static property	Returns a new **RegKey** object pointing to HKCC key.
HKEY_DYN_DATA	static property	Returns new **RegKey** object pointing to HKDD key.

Example

The following script determines whether **HKCR** is valid and retrieves some values from the **HKCU\Sofrware\mykey** key.

```
try
  {
  var  rk  =  new  RegKey();
  rk.Connect(RegKey.HKEY_CURRENT_USER,  "server");
  rk.Open(rk,  "Software\\mykey");
  print("string="  +  rk.GetString("StringValue"));
  print("dword="  +  rk.GetDWORD("DWORDValue"));
  print("multistring=");
  var  ms  =  rk.GetMultiString("Multiline  string  value");
  for  (x  in  ms)  {
  print(ms[x]);
  }
  }
  catch  (err)
  {
  print("Error:  "  +  err);
  }
```

ScriptExecObject

NAME	TYPE	DESCRIPTION
StdErr	property	Provides access to the stderr output stream of the object. Read-only.
StdOut	property	Provides access to the stdout output stream of the object. Read-only.
Wait	method	Waits until process return.
Terminate	method	Ungracefully terminates the process.

ShellObject

NAME	TYPE	DESCRIPTION
ExpandEnvironmentStrings (strString)	method	Returns an environment variable's expanded value. Variables in **strString** should use the % character.
Exec(strCommand)	method	Unix only. Runs an application in a child command-shell, providing access to the **StdOut** and **StdErr** streams. The **Exec** method returns a ScriptExecObject, which provides status

NAME	TYPE	DESCRIPTION
		and error information about a script run with **Exec** along with access to the **StdOut** and **StdErr** channels.
Read(arrStreams, nCharCount)	static method	Reads a number of characters specified by **nCharCount** from any available streams in **arrStreams**. Also sets a **Signaled** property of the **arrStreams** array to the index of a signaled stream.
ReadLine(arrStreams)	static method	Reads a line from any available streams in **arrStreams**. Also sets a **Signaled** property of the **arrStreams** array to the index of a signaled stream.
AtEndOfAllStreams (arrStreams)	static method	Checks whether all streams in the **arrStreams** array are at the end of the stream.

Example

```
var sh = new ShellObject();
 print(sh.ExpandEnvironmentStrings("PATH=%PATH%"));
 var ls = sh.Exec("ls -l");
 var out = ls.StdOut;
 while (!out.AtEndOfStream) {
 s = out.ReadLine();
 print( s+"\n" );
```

SystemInformationObject

NAME	DESCRIPTION
GetComputerName (address)	Retrieves the NetBIOS or DNS name associated with the specified computer address. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
GetDomainName (address)	Retrieves the domain name. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
GetComputerRole (address)	Retrieves the computer role. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
GetPlatformId (address)	Retrieves the platform ID. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
GetVersionMajor (address)	Retrieves the major number of the operating system version of the computer with the specified address. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.

NAME	DESCRIPTION
GetVersionMinor (address)	Retrieves the minor number of the operating system version of the computer with the specified address. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
GetTimeOfDayInfo (address)	Returns a TimeOfDayInfoObject object. The address parameter is a NetBIOS name. If the parameter is not specified, the method works with the local computer.
IsLocalHost (address)	Verifies whether the specified address (or computer name) is the address of the local computer. The address parameter is a NetBIOS name or an IP address. If the parameter is not specified, the method works with the local computer.

Example

The following script displays all available information about the "SERVER" computer.

```
try
  {
  var  si  =  new  SystemInformationObject();
  print(si.GetComputerName("server"));
  print(si.GetDomainName("server"));
  print(si.GetPlatformId("server"));
  print(si.GetVersionMajor("server"));
  print(si.GetVersionMinor("server"));
  print(si.IsLocalHost("server"));

  var  ti  =  si.GetTimeOfDayInfo("server");
  print(ti.elapsedt);
  print(ti.msecs);
  print(ti.hours);
  print(ti.mins);
  print(ti.secs);
  print(ti.hunds);
  print(ti.timezone);
  print(ti.tinterval);
  print(ti.day);
  print(ti.month);
  print(ti.year);
  print(ti.weekday);
  }
  catch  (err)
  {
  print("Error:  "  +  err);
  }
```

TextStream

NAME	TYPE	DESCRIPTION
Read([count])	method	Returns a single character (by default) or the specified number of characters according to the current encoding.
ReadLine()	method	Gets a line from the file.
Close()	method	Closes the file. If you do not close the file explicitly, it will be automatically closed during the next garbage collection session.
AtEndOfStream	property	Returns true if the file pointer is at the end of a **TextStream** file; false otherwise. Read-only.
TellG	property	Returns the number of bytes read from the stream and dispatched into the script. Not aligned to internal buffer boundary.

TimeOfDayInfoObject

NAME	DESCRIPTION
elapsedt	Specifies the number of seconds since 00:00:00, January 1, 1970, GMT.
msecs	Specifies the number of milliseconds from an arbitrary starting point (system reset).
hours	Specifies the current hour. Valid values are 0 through 23.
mins	Specifies the current minute. Valid values are 0 through 59.
secs	Specifies the current second. Valid values are 0 through 59.
hunds	Specifies the current hundredth of a second. Valid values are 0 through 99.
timezone	Specifies the time zone of the server. This value is calculated, in minutes, from Greenwich Mean Time (GMT). For time zones west of Greenwich, the value is positive; for time zones east of Greenwich, the value is negative. A value of -1 indicates that the time zone is undefined.
tinterval	Specifies the time interval for each tick of the clock. Each integral integer represents one ten-thousandth of a second (0.0001 second). day Specifies the day of the month. Valid values are 1 through 31.
month	Specifies the month of the year. Valid values are 1 through 12.
year	Specifies the year.
weekday	Specifies the day of the week. Valid values are 0 through 6, where 0 is Sunday, 1 is Monday, and so on.

See also the example for the **SystemInformationObject** object.

InTrust-Specific Objects

Glossary

An InTrust *organization parameter* is a setting that affects some aspect of InTrust operation related to agents, jobs, licenses and so on. In InTrust Manager, these settings can be configured in the properties of the InTrust root node, an InTrust server or an agent.

A *facet* is a file with an agent's organization parameter settings. An agent has one facet per InTrust server; if an agent responds to multiple servers, it has multiple facets. The file is located in the **config** subfolder of the agent installation folder, and the file name is the GUID of the server from which the agent inherits organization parameter settings.

A facet can contain organization parameter values defined at the following three levels:

1. Organization level, if any parameters are inherited all the way down
2. Server level, if any parameters are defined for the InTrust server and inherited by the agent
3. Agent level, if any parameters are defined specifically for the agent

Organization parameter settings can be read using the **ADCEnvironment** object. See Organization Parameter Editor for details about setting parameter values for organizations, servers and agents.

Generic Objects

- ScriptState
- GlobalState
- Reference
- ADCEnvironment

Audit Script Object Model

The following objects are available to scripts in custom text log data sources. In addition, these data sources can use all objects available to custom scripted data sources.

- GlobalObject
- AuditProvider
- Event
- Position
- ErrorInfo

Script Callbacks

Provider callbacks

NAME	DESCRIPTION
Audit_ EnumInstances (host)	Returns a two-dimensional string array, where the nested arrays have two elements. Element [i][0] is the name of the i-th instance and element [i][1] is a display name. For Windows computers, the display name is the NetBIOS name. For Unix computers, it is the localhost name. The **host** parameter can be one of the following: • Empty string for localhost • Host name • IP address
Audit_ BeforeCollection	No return value.
Audit_Connect (instance, log)	Boolean.
Audit_Seek (position)	Boolean. A Position object is passed to this function. **Audit_Seek** uses the object to determine which event gathering stopped at when data was last collected. If the precise position is found, true is returned, otherwise false is returned. If the result is false, InTrust session details will show that the position cannot be found.
Audit_ CollectEvents()	Returns an object which is a collection of user-supplied properties. Performs log gathering.
Audit_ AfterCollection (success)	No return value.

Comparer callback

NAME	DESCRIPTION
Audit_ComparePositions(p1, p2)	Returns a signed integer indicating the ordering of **p1** and **p2**. • >0 if **p1** > **p2** • 0 if **p1** == **p2** • <0 if **p1** < **p2** where **p1** and **p2** are **Position** objects.

Audit Script Engine and Audit Script Host Cooperation

The audit script engine exposes a C++ interface to ECMAScript.
This interface does the following:

- Supports the scripting object model
- Dispatches calls between the audit script host and the scripting machine
- Converts parameters from C++ types to scripting types and vice versa

Type conversion takes place as follows:

- C++ type to script callback:

CONVERTED FROM	CONVERTED TO
AuditScriptEngine::EnumInstances	script::Audit_EnumInstances
AuditScriptEngine::GetDataName	script:: Audit_GetDataName
AuditScriptEngine::BeforeCollection	script:: Audit_BeforeCollection
AuditScriptEngine::Connect	script:: Audit_Connect
AuditScriptEngine::Seek	script:: Audit_Seek
AuditScriptEngine::CollectEvents	script:: Audit_CollectEvents
AuditScriptEngine::AfterCollection	script:: Audit_AfterCollection

- Script object model to C++ type:

CONVERTED FROM	CONVERTED TO
AuditProvider::SubmitEventPositionPair	AuditScriptHost:: SubmitEventPositionPair
AuditProvider::LogMessage	AuditScriptHost::LogMessage
AuditProvider::OperationStatus	AuditScriptHost::OperationStatus
AuditProvider::Trace	AuditScriptHost::Trace

Audit Script Position Processor and Audit Script Host Cooperation

Audit script position processor is a middle layer between Audit script host and position comparer script. Audit script position processor should dispatch calls from AuditScriptPositionProcessor::ComparePositions to the script's Audit_ComparePositions callback.

Error Reporting

Audit script engine errors can be reported in two ways:

- Using the **ErrorInfo** object (the result is written to the InTrust Server log)
- As exceptions of the **ADCException** type for the audit script host to catch (the result is included in the InTrust session information)

Notes about exceptions:

- Scripting callbacks should raise a JScript exception if they encounter any exceptional situation.
- The audit script engine should catch such exceptions, translate them to **ADCException** objects and throw them again as C++ exceptions.
- Script exceptions are translated into C++ exceptions. Exception information appears in session details.
- The audit script position processor has the same error handling model as the audit script engine.

ADCEnvironment

The object is used for reading InTrust organization parameter settings from agents. This object is available only with the ECMAScript object model.

Methods

NAME	DESCRIPTION
EnumFacets()	Enumerates the facets available for an agent and returns an array of facet names.
GetEnvironmentValue ([Facet,] Name)	Returns the value of the specified organization parameter. The optional Facet string argument is the name of a facet, as returned by the **EnumFacets()** method. If this argument is not specified, the method returns the organization parameter value from the first available facet. The **Name** string argument is the name of the organization parameter value.
GetEnvValueWithDefault ([Facet,] Name, DefValue)	Returns the value of the specified organization parameter. This method is different from **GetEnvironmentValue()** in that you can specify a default return value with the **DefValue** argument. This default value is returned if the specified organization parameter is not found, so you do not have to perform any exception checks in your code. The optional **Facet** string argument is the name of a facet, as returned by the **EnumFacets()** method. If this argument is not specified, the method returns the organization parameter value from the first available facet. The **Name** string argument is the name of the organization parameter value. The **DefValue** argument is a string or an integer.
ExpandEnvironmentString (String)	This method expands the name of the supplied InTrust-specific environment variable, such as %ADC_INSTALL_PATH% or %ADC_DATA_PATH%.

Example

Running the following script in a single-server InTrust organization:

```
var objEnv = new ADCEnvironment();
 var Facets = objEnv.EnumFacets();
 for(var i = 0; i < Facets.length; i++)
 {
 var cputhrottling = objEnv.GetEnvValueWithDefault(Facets[i], "ADC_
AgentCPUThrottlingValue", 100);
 Trace(40, "Facet: " + Facets[i]);
 Trace(40, "CPU: " + cputhrottling);
 }
 var datapath = objEnv.GetEnvironmentValue("adc_data_path");
 Trace(40, "Data Path: " + datapath);
 var taskpath = objEnv.ExpandEnvironmentString(datapath + "\\tasks");
 Trace(40, "Tasks Path: " + taskpath);
```

will produce results similar to the following in the trace file:

(392) Thu_Aug_23_21.12.32.437_2007 | 40 | [027035d8] ScriptContext_SM: Facet: {ae146e4d-9fc4-47ae-97e0-bf5cb55b0890} (string)

(392) Thu_Aug_23_21.12.32.437_2007 | 40 | [027035d8] ScriptContext_SM: CPU: 100 (string)

(392) Thu_Aug_23_21.12.32.437_2007 | 40 | [027035d8] ScriptContext_SM: Data Path: %ADC_INSTALL_PATH%/data (string)

(392) Thu_Aug_23_21.12.32.437_2007 | 40 | [027035d8] ScriptContext_SM: Tasks Path: C:\WINNT\ADCAgent/data\tasks (string)

AuditProvider

Non-creatable.

NAME	TYPE	DESCRIPTION
SubmitEventPositionPair(event, position)	method	Submits event and position pair to the audit provider.
LogMessage(messagetext, systemmessage, messageseverity)	method	**messageseverity** is an integer: 4 = success 3 = information 2 = error 1 = warning
OperationStatus(statustext)	method	**Trace(level, msg)** method recommended level values: 0 = TraceDisabled 10 = TraceCritical 20 = TraceWarning 30 = TraceNormal 40 = TraceDebug

Example

```
AuditProvider.SubmitEventPositionPair(new Event(), new Position());
```

ErrorInfo

This object lets you provide detailed error messages in the InTrust Server log. The object is available only in the object model of audit scripts based on JScript or VBScript.

NAME	TYPE	DESCRIPTION
Number	property	Event ID of the error.
Description	property	Error description.
Raise	method	Submits the error to the InTrust server log; this method has no parameters.

Example

This snippet puts an error event in the InTrust Server log. The message description contains the string "Invalid password".

```
ErrorInfo.Description = "Invalid password";
ErrorInfo.Raise();
```

Event

Creatable.

This object is a collection of user-supplied properties.

Example

```
x = new Event;
x.prop1 = "val1";
x.prop2 = new Date()
```

GlobalObject

Non-creatable.

NAME	TYPE	DESCRIPTION
AuditProvider	property	Returns an AuditProvider object.

GlobalState

The **GlobalState** object is similar to the **ScriptState** object, but with the following differences:

- It is available only in JScript.
- It stores the state of the object only as long as the agent process is active.
- It can store objects of any type.

Example

```
if ( !GlobalState.Item("shell") )
 {
 GlobalState.Item("shell") = new ActiveXObject("WScript.Shell");
 }
 #TRACE 40 GlobalState.Item("shell").Value.ExpandEnvironmentStrings("%TMP%")
```

Position

Creatable.

NAME	TYPE	DESCRIPTION
RecordKey	property	Has integral type
Time	property	Has Date type
Values	property	Collection of user-supplied properties

Example

```
x = new Position;
 x.RecordKey = 12;
 x.Time = new Date();
 x.Values.prop1 = "val1";
 x.Values.prop2 = new Date();
```

Reference

The **Reference** object is a wrapper used mainly for passing the out parameter to COM methods. To create a call, use the function **CreateReference(value)** where value is an optional argument that specifies the start value. This object is available when you use the ECMAScript or JScript object model.

You cannot correctly reference a property if at least one of its parameters is [in, out].

IDL example

```
HRESULT Increment([in, out] LONG * param);
```

JScript example

```
var ref = CreateReference(5);
 var obj = new ActiveXObject("MyObject");
 obj.Increment(ref);
 x = ref; // x = 6;
 ECMAScript example:
 var ref = CreateReference(5);
 var obj = new ActiveXObject("MyObject");
 DispInvoke(obj, "Increment", ref);
 x = ref.RefOn; // x = 6;
```

ScriptState

ScriptState is an object with arbitrary fields that stores values between calls and between InTrust agent restarts. This object is available when you use the ECMAScript or JScript object model.

The **ScriptState** object can store only the following data types:

- Strings
- Numeric types
- Dates
- Arrays of all of the above

GlobalState is a similar object that can store any data type, but is available only in the JScript object model.

In the following two examples, every fifth event is matched.

ECMAScript Example

```
function  OnEvent(  e  )
  {
  if  (ScriptState.count  ==  5  )
  {
  match(  e  );
  ScriptState.count  =  0;
  }
  else
  {
  ScriptState.count++;
  }
  }
```

JScript Example

```
function  OnEvent(  e  )
  {
  if  (ScriptState.Item("count")  ==  5  )
  {
  match(  e  );
  ScriptState.Item("count")  =  0;
  }
  else
  {
  ScriptState.Item("count")++;
  }
  }
```

We are more than just a name

We are on a quest to make your information technology work harder for you. That is why we build community-driven software solutions that help you spend less time on IT administration and more time on business innovation. We help you modernize your data center, get you to the cloud quicker and provide the expertise, security and accessibility you need to grow your data-driven business. Combined with Quest's invitation to the global community to be a part of its innovation, and our firm commitment to ensuring customer satisfaction, we continue to deliver solutions that have a real impact on our customers today and leave a legacy we are proud of. We are challenging the status quo by transforming into a new software company. And as your partner, we work tirelessly to make sure your information technology is designed for you and by you. This is our mission, and we are in this together. Welcome to a new Quest. You are invited to Join the Innovation™.

Our brand, our vision. Together.

Our logo reflects our story: innovation, community and support. An important part of this story begins with the letter Q. It is a perfect circle, representing our commitment to technological precision and strength. The space in the Q itself symbolizes our need to add the missing piece — you — to the community, to the new Quest.

Contacting Quest

For sales or other inquiries, visit https://www.quest.com/company/contact-us.aspx or call +1-949-754-8000.

Technical support resources

Technical support is available to Quest customers with a valid maintenance contract and customers who have trial versions. You can access the Quest Support Portal at https://support.quest.com.

The Support Portal provides self-help tools you can use to solve problems quickly and independently, 24 hours a day, 365 days a year. The Support Portal enables you to:

- Submit and manage a Service Request
- View Knowledge Base articles
- Sign up for product notifications
- Download software and technical documentation
- View how-to-videos
- Engage in community discussions
- Chat with support engineers online
- View services to assist you with your product