A10 Networks

P e r f o r m a n c e   b y   D e s i g n

EX Series Secure WAN Manager aFleX Scripting Language Reference

Document No.: D-020-01-00-0008

aFleX Engine Ver. 2.0 3/5/2010

Headquarters

A10 Networks, Inc.
2309 Bering Dr.
San Jose, CA 95131-1125 USA

Tel: +1-408-325-8668 (main)
Tel: +1-408-325-8676 (support - worldwide)
Tel: +1-888-822-7210 (support - toll-free in USA)
Fax: +1-408-325-8666

www.a10networks.com

Published by:
A10 Networks, Inc.
2309 Bering Dr.
San Jose, CA 95131-1125
USA

Disclaimer

The information presented in this document describes the specific products noted and does not imply nor grant a guarantee of any technical performance nor does it provide cause for any eventual claims resulting from the use or misuse of the products described herein or errors and/or omissions. A10 Networks, Inc. reserves the right to make technical and other changes to their products and documents at any time and without prior notification.

No warranty is expressed or implied; including and not limited to warranties of non-infringement, regarding programs, circuitry, descriptions and illustrations herein.

Environmental Considerations

Some electronic components may possibly contain dangerous substances. For information on specific component types, please contact the manufacturer of that component. Always consult local authorities for regulations regarding proper disposal of electronic components in your area.

Further Information

For additional information about A10 products, terms and conditions of delivery, and pricing, contact your nearest A10 Networks, Inc. location which can be found by visiting www.a10networks.com.

About This Document

This document describes the aFleX inline scripting engine and aFleX Policy Editor, used with the A10 Networks EX Series Secure WAN Manager.

Note: The commands and options described in this edition are supported with EX Release 3.0 or later.

Additional information is available for EX Series systems in the following documents. These documents are included on the documentation CD shipped with your EX Series system, and also are available on the A10 Networks support site:

- *EX Series Installation and Setup Guide*
- *EX Series Command Line Interface User Manual*
- *EX Series Graphical User Interface User Manual*
- *EX Series aFleX Reference*

Audience

This document is intended for use by system administrators for provision and maintenance of the A10 Networks EX Series; specifically for reference in authoring and implementing aFleX policy scripts.

aFleX Basics

Overview

The aFleX scripting language is a powerful inline custom scripting engine that provides Layer 7 inspection and classification of traffic.

The aFleX scripting language is based on the Tool Command Language (Tcl) programming standard for simplicity and familiarity.

Example aFleX Script

The following script is activated when the EX device receives a data packet from a client (CLIENT_DATA event).

```
when CLIENT_DATA {
    if { [IP::addr [IP::client_addr] equals 192.168.3.39] } {
        matchaflex
        return
    } elseif { [IP::addr [IP::server_addr] equals 192.168.3.39] } {
        matchaflex
        return
    } elseif { [TCP::payload] contains "a10networks.com"} {
        matchaflex
        return
    }
    if { [UDP::payload] starts_with "OK" } {
        matchaflex
        return
    }
}
```

The commands in the script classify traffic that meets any of the following conditions:

- The client IP address is 192.168.3.39.
- The server IP address is 192.168.3.39
- The packet has a TCP payload that contains the string “a10networks.com”.
- The packet has a UDP payload that starts with the string “OK”.

For matching traffic, the matchaflex command performs Layer 7 classification of the traffic. The return command then returns to the script.

The following script is activated when the EX device receives a data packet from a server (SERVER_DATA event). If the IP protocol number of the packet is 50, the matchaflex command performs Layer 7 classification of the traffic. The return command then returns to the script.

```
when SERVER_DATA {
    if { [IP::protocol] == 50 } {
        matchaflex
        return
    }
}
```

The following script performs Layer 7 classification of multiple packets.

```
when SERVER_DATA {
    if { [L7CL::pkts rev] == 1 } {
        if { ([L7CL::get_flag 1] == 1) and ([TCP::payload] starts_with "HTTP")
} {
            matchaflex
            return
        }
    }
}

when CLIENT_DATA {
    if { [L7CL::pkts fwd] == 1 } {
        if { [TCP::payload] contains "GET " } {
            L7CL::set_flag 1
            return
        }
    }
}
```

When aFleX Script Changes Take Effect

Changes to aFleX scripts affect only new sessions or connections that are created after the script change is implemented. Sessions or connections that are already active when a script is changed are not affected by the script change.

Maximum File Size of aFleX Scripts

By default, the maximum file size supported on an EX device for an aFleX script is 32 Kbytes. On the EX device, you can change the maximum script size, to 16-128 Kbytes.

To change the maximum aFleX file size, use the following command at the global configuration level of the CLI:

```
[no] aflex max-filesize KBytes
```

aFleX Syntax

An aFleX script is a Tcl-like script.

Tcl Symbols

The Tcl symbols listed in Table 1 have special meanings.

TABLE 1 Tcl Symbols Supported in aFleX Policies

Delimiter	Description
`$`	Variable substitution. Example: `$argv0` could be replaced by `/usr/bin/somescript.tcl`
`[ ]`	Subcommand substitution. Example: `[pwd]` could be replaced by `/home/joe`
`" "`	Word grouping with substitutions. Example `"you are $user"` is one word. Substitution still occurs.
`{ }`	Word grouping without substitutions. Example: {you are `$user`} is one word. `$user` is not replaced.
`\`	Backslash substitution/escape or statement continuation. By default, a statement ends with the end of the line.
#	Comment. This symbol can be used only at the beginning of a statement.
`;`	Statement separator.
`: :`	Namespace path separator for variables or commands. Example: `::foo::bar`

For information about standard Tcl syntax, see the following:

http://tmml.sourceforge.net/doc/tcl/index.html

http://en.wikibooks.org/wiki/Programming:Tcl

aFleX Script Components

aFleX scripts consist of the following element types:

- Events
- Operators
- Commands

aFleX Events

aFleX scripts are event-driven. The EX device triggers an aFleX policy based on a specified event. For example, if an aFleX policy is configured to be triggered by the HTTP_REQUEST event, the EX device triggers the aFleX policy when an HTTP request is received.

Event declarations are made with the "when" keyword followed by the event name. Here is an example:

```
when CLIENT_DATA {
     if { [IP::addr [IP::remote_addr] equals 10.1.1.80 ] } {
          matchaflex
          return
     }
}
```

The script in this example is triggered when a client data packet is received (CLIENT_DATA event). If the client IP address is 10.1.1.80, the matchaflex command is executed, to perform Layer 7 classification of the traffic. The script is then terminated, by the return command. Table 2 lists the event declarations supported in aFleX policies.

TABLE 2 aFleX Event Declarations

Event Type	Event Name and Description
Global	`RULE_INIT` Triggered when used in an aFleX policy.
IP, TCP, UDP	`CLIENT_DATA` Triggered when data is received from a client during Layer 7 classification of a new connection.
	`SERVER_DATA` Triggered when data is received from a server during Layer 7 classification of a new connection.

aFleX Operators

aFleX policies use operators to compare operands in an expression.

Table 3 lists the operators supported in aFleX policies.

TABLE 3 aFleX Operators

Operator Type	Operator Name and Description
Relational	`contains` Tests whether one string (string1) contains another string (string2).
	`ends_with` Tests whether one string (string1) ends with another string (string2).
	`equals` Tests whether one string equals another string.
	`matches` Tests whether one string matches another string.
	`matches_regex` Tests whether one string matches a regular expression.
	`starts_with` Tests whether one string (string1) starts with another string (string2).
	`switch` Built-in Tcl command. Evaluates one of several scripts, depending on a given value.
Logical	`and` Performs a logical "and" comparison between two values.
	`or` Performs a logical "or" comparison between two values.
	`not` Performs a logical "not" on a value.

aFleX Command Summary

Table 4 lists the aFleX commands according to the types of operations they perform. For more information about the aFleX commands, see "Command Reference" on page 25, where they are listed alphabetically.

TABLE 4 aFleX Commands

Command Type	Command Name and Description
Global	`b64decode <string>` Returns the specified string, decoded from base-64. Returns NULL if there is an error.
	`b64encode <string>` Returns the specified string, encoded as base-64. Returns NULL if there is an error.
	`clientside {<aFleX commands>}` Causes the specified aFleX commands to be evaluated under the client-side context. This command has no effect if the aFleX policy is already being evaluated under the client-side context.
	`event [<name>] [enable | disable] | [enable all | disable all]` Discontinues evaluating the specified aFleX event, or all aFleX events, on a connection. However, the aFleX script continues to run.
	`htonl <hostlong>` Converts the unsigned integer from host byte order to network byte order.
	`htons <hostshort>` Converts the unsigned short integer from host byte order to network byte order.

	`if { <expression> } {<statement_command>}`
	`elseif { <expression> } {<statement_command>}`
	Asks a true or false question and, depending on the answer, takes some action.
	Note: The maximum number of `if` statements that you can nest in an aFleX policy is 100.

	`matchaflex` Matches an aFleX rule for Layer 7 classification.
	`ntohl <netlong>` Converts the unsigned integer from network byte order to host byte order.
	`ntohs <netshort>` Converts the unsigned short integer from network byte order to host byte order.
	`return [<expression>]` Terminates execution of the aFleX event and optionally return the result of evaluating <expression>.
	`serverside {<aFleX commands>}` Causes the specified aFleX commands to be evaluated under the server-side context. This command has no effect if the aFleX policy is already being evaluated under the server-side context.
	`when <event_name>` Specify an event in an aFleX script. All aFleX events begin with a `when` command. You can specify multiple `when` commands within a single aFleX script.

TABLE 4 aFleX Commands (Continued)

Command Type	Command Name and Description
IP Packet Header Query	`IP::addr <addr1>[/<mask>] equals <addr2>[/<mask>]` Performs comparison of IP address/subnet/supernet to IP address/subnet/supernet. Returns 0 if no match, 1 for a match.
	`IP::client_addr` Returns the client IP address of a connection. This command is equivalent to the command clientside { IP::remote_addr }.
	`IP::local_addr` Returns the local IP address of a connection.
	`IP::protocol` Returns the IP protocol value.
	`IP::remote_addr` Returns the remote IP address of a connection.
	`IP::server_addr` Returns the server's IP address. This command is equivalent to the command serverside { IP::remote_addr }.
	`IP::stats {pkts in | pkts out | pkts | bytes in | bytes out | bytes | age}` Supplies information about the number of packets or bytes being sent or received in a given connection.
	`IP::tos` Returns the value of the IP protocol's Type of Service (ToS) field.
	`IP::ttl` Returns the TTL of the current packet being acted upon.
	`IP::version` Return the version (e.g., IPv4/IPv6) of the current packet.
Layer 7 Classification	`L7CL::get_flag <offset>` Returns the Layer 7 classification flags, by offset.
	`L7CL::pkt_cnt [<fwd> | <rev>]` Returns the count of packets that have been Layer-7 classified. The count includes the current packet.
	`L7CL::set_flag <offset>` Sets the Layer 7 classification flags, by offset.

TABLE 4 aFleX Commands (Continued)

Command Type	Command Name and Description
TCP Packet Header and Content Query	`TCP::client_port` Returns the client's TCP port/service number. Equivalent to the command `clientside { TCP::remote_port }`.
	`TCP::local_port` Returns the local TCP port/service number.
	`TCP::mss` Returns the on-wire Maximum Segment Size (MSS) for a TCP connection.
	`TCP::payload [<size>]` Returns the accumulated TCP data content.
	`TCP::payload length` Returns the TCP data length of the packet.
	`TCP::remote_port` Returns the remote TCP port/service number.
	`TCP::server_port` Returns the server TCP port/service number. Equivalent to the command `serverside { TCP::remote_port }`.
UDP Packet Header and Content Query	`UDP::client_port` Returns the client's UDP port/service number. **Note:** This command is equivalent to the command `clientside {UDP::remote_port}`.
	`UDP::local_port` Returns the local UDP port/service number.
	`UDP::mss` Returns the on-wire Maximum Segment Size (MSS) for a UDP connection.
	`UDP::payload [<size>]` Returns the current UDP payload content.
	`UDP::payload length` Returns the amount of UDP payload content in bytes.
	`UDP::remote_port` Returns the remote's UDP port/service number.
	`UDP::server_port` Returns the server UDP port/service number. **Note:** This command is equivalent to the command `serverside { UDP::remote_port }`.

TABLE 4 aFleX Commands (Continued)

Command Type	Command Name and Description
Deep packet inspection	`findstr` Finds the string <search_string> within <string> and returns a sub-string based on the <skip_count> and <terminator> from the matched location.
	`getfield` Splits a string on a character, and returns the string corresponding to the specific field.
	`substr` Returns a sub-string <string> based on the values of <skip_count> and <terminator>.
	`domain` Parses the string <string> as a dotted domain name and return the last <count> portions of the domain name.

Disabled Tcl Commands

For security, the following Tcl commands are disabled in the aFleX syntax. You cannot use these commands in aFleX scripts.

after	exec	interp	seek
auto_execok	exit	load	socket
auto_import	fblocked	memory	source
auto_load	fconfigure	namespace	tcl_findLibrary
auto_mkindex	fcopy	open	tell
auto_mkindex_old	file	package	unknown
auto_qualify	fileevent	pid	update
auto_reset	filename	pkg::create	uplevel
bgerror	flush	pkg_mkIndex	upvar
cd	gets	proc	vwait
close	glob	pwd	
eof	http	rename	

aFleX Context – Clientside or Serverside

aFleX scripts support context for specifying either client or server side:

- Each event has a default context of either client-side or server-side.
- Key words: “clientside” or “serverside”
- Only specify the context keywords if you want to change default context.

Example:

The following script uses a client-side event:

```
when CLIENT_DATA {
     if { [IP::addr [clientside {IP::remote_addr}] equals 192.168.3.252] } {
     matchaflex
     return
     }
}
```

The following script uses a server-side event:

```
when SERVER_DATA {
     if { [IP::addr [clientside {IP::remote_addr}] equals 192.168.3.252] } {
     matchaflex
     return
     }
}
```

aFleX Library

The EX Series documentation CD includes a library of aFleX policies. You can use these policies as is or copy and modify them. Table 5 lists the aFleX policies included in the “aFleX Library” folder on the documentation CD.

TABLE 5 aFleX Library

Script	Description
360download.afx	Matches on payload information when a client uses the 360downloader tool.
client_addr.afx	Matches on client IP address.
client_port.afx	Matches on a specific client-side protocol port. The sample script matches on port 23.
clientside.afx	Matches on a specific client IP address. The sample script matches on client IP address 192.168.3.252.
IP_Protocol.afx	Matches on a specific IP protocol. The sample script matches on IP protocol 6.
local_addr.afx	Matches on a specific local IP address. The sample script matches on client IP address 192.168.3.252.

TABLE 5 aFleX Library (Continued)

Script	Description
openv.afx	Matches on OpenV video traffic.
QvodPlayer.afx	Matches on QvodPlayer P2P traffic.
remote_port serverside.afx	Matches on the remote protocol port in server data traffic.
server_addr.afx	Matches on server IP address in client data traffic.
server_port bothside.afx	Matches on server protocol port in client-side and server-side traffic.
serverside.afx	Matches the remote address of server-side traffic, for client and server traffic.
shareaza.afx	Matches on Layer 7 payload characters in Shareaza P2P traffic.
sinatv.afx	Matches on Layer 7 payload characters in SinaTV online video traffic.
udp-client_port means sport.afx	Matches on client UDP port. The sample script matches on client UDP port 69.
udp-payload_bin.afx	Translates binary characters for matching UDP::payload special characters.

Importing and Binding aFleX Scripts

To use an aFleX policy:

1. Import the aFleX policy onto the EX device.

2. Bind the aFleX policy to a match rule for a class. Layer 7 traffic that matches the aFleX policy will be added to the class.

The following sections show examples for the CLI and GUI.

Using the CLI

1. On a PC that supports TFTP, FTP, SCP or RCP, use any text editor to create an aFleX script and save it locally. Use extension ".afx" at the end of the file name. Example: /aflex/test.afx

2. On the EX device, use the CLI command **aflex import** to import the aFleX policy file onto the EX device.

3. Use the CLI command **match aflex** under class match rule configuration.

CLI Example

This example shows how to import an aFleX policy onto the EX device and add it to a match rule for a class.

1. Log onto the EX device through the CLI, with an admin account that has read-write privileges.

   A CLI prompt appears:

   ```
   EX>
   ```

Note: See the *EX Series CLI Reference* if you need information on using the CLI.

2. Access the Privileged EXEC mode:

```
EX>enable
Password:***
EX#
```

3. Access the configuration mode:

```
EX#config
EX(config)#
```

4. Use the **aflex import** command to import the aFleX policy (“http-manual.afx”) onto the EX device:

```
EX(config)#aflex import http-manual scp://192.168.1.118/aflex/http-manual.afx
User name []?***
Password []?***
Importing ... Done.
EX(config)#
```

While importing the aFleX policy, the EX device checks for syntax errors. If any syntax errors are found, error messages are displayed. You can modify an aFleX policy and import it again until it passes the syntax check.

5. Use the **show aflex** command to list the aFleX policies imported onto the EX device:

```
EX(config)#show aflex
Total aFlex number: 1
Max aFlex file size: 32K
Name                             Syntax   Qos class
------------------------------------------------------------
http-manual                      Check    No
```

Note: To display the aFleX policy itself, use the **show aflex** *aflex-name* command.

6. Access the configuration mode for a QoS class.

```
EX(config)#qos class http-manual category Application
EX(config-class)#
```

7. Bind the aFleX policy to a match rule for the class, and verify the binding.

```
EX(config-class)#match aflex http-manual
EX(config-class)#show this

qos class http-manual              category Application   (None)
   match (0) aflex http-manual
```

8. Show the aFleX policy list again. The output indicates that the syntax of the aFleX script has been checked and the policy is bound to a class rule.

```
EX(config)#show aflex
Total aFlex number: 1
Max aFlex file size: 32K
Name                             Syntax   Qos class
------------------------------------------------------------
http-manual                      Check    Bind
```

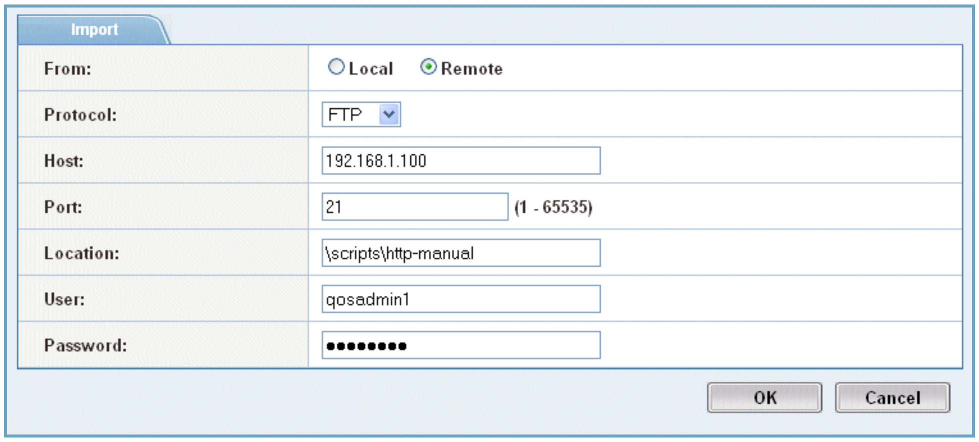
Using the GUI

Import or Create the aFleX Script

1. Select Config > QoS > Class.
2. On the menu bar, select aFleX.
3. To import an aFleX script configured on another device, click Import and go to step 4. Otherwise, to enter the script configuration on the GUI, click New and go to step 5.
4. Importing an aFeX script:
   a. Select the location of the script:
   - Local – The script is on the PC from which you are accessing the GUI. Click Browse, navigate to the file, and click Open.
   - Remote – The script is on another device. Select the file transfer protocol, enter access information for the remote device, and the location (path and filename), and click OK.
5. Entering a script in the GUI:
   a. Enter a name for the aFleX script in the Name field.
   b. Enter the aFleX policy text into the Definition field.
   c. Click OK.

Note: You can click on the name of an existing aFleX policy to edit it in the GUI.

FIGURE 1 *Config > QoS > Class > aFleX - Import*

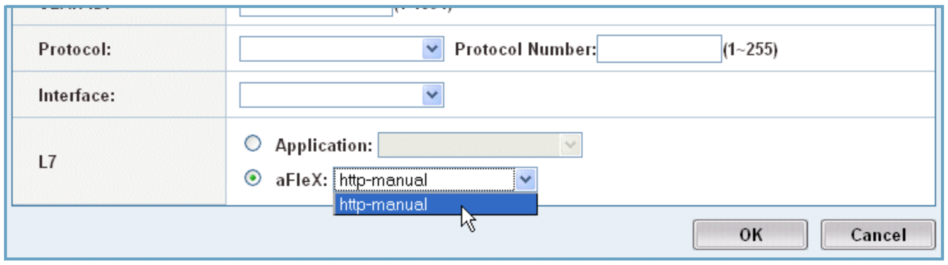


From:	Local [✓ Remote]
Protocol:	FTP
Host:	192.168.1.100
Port:	21 (1 - 65535)
Location:	\scripts\http-manual
User:	qosadmin1
Password:	●●●●●●●●●

Bind the Script to a Match Rule for a QoS Class

1. Select Config > QoS > Class.
2. On the menu bar, select Class, if not already selected.
3. Click on the class name to configure a rule for an existing class, or click New at the bottom of the page to create a new class.
4. Click New at the right of the Rules List.
5. Select aFleX next to L7, and select the aFleX policy from the drop-down list.
6. Enter or select other rule settings if applicable.
7. When finished configuring the rule, click OK. The rule appears in the Rules List for the class.

FIGURE 2 *Config > QoS > Class - match rule configuration*



Protocol:	Protocol Number: (1~255)
Interface:	
L7	Application: [✓ aFleX: http-manual]

Command Reference

aFleX scripts consist of three basic elements:

- Events
- “Operators” on page 27
- “Commands” on page 35

These elements are described in detail in subsequent sections.

Events

The following subsections describe the aFleX events.

Global Events

RULE_INIT

Initializes global system variables. Within an aFleX policy, the RULE_INIT event can initialize a system variable on a global basis for all aFleX policies, or exclusively for that particular aFleX policy.

The prefix placed before RULE_INIT specifies whether to initialize the variable for all aFleX policies, or only the current aFleX policy.

Prefix	Scope
`::`	Applies only to the current aFleX policy. This variable cannot be set or read by any other aFleX policies. Once the variable is defined, it can be removed only by an unset command.
`::global::`	Applies to all aFleX policies. This variable can be set or read by all aFleX policies on the EX device.

Note: Unbinding an aFleX policy will not remove the variable.

Example

```
when RULE_INIT {
# define per-aFleX global variable ::request_count
# This variable is to count the # of CLIENT_DATA hits by this aFleX policy
     set ::request_count 0
# define per-system global variable ::global::ax_request_count
# This variable is to count the total number of CLIENT_DATA hits
# in the EX system
     set ::global::ex_request_count 0
# Remove per aFleX global variable ::remove_var1
     unset ::remove_var1
}
when CLIENT_DATA {
     incr ::request_count
     incr ::global::ax_request_count
}
```

IP, TCP, and UDP Events

CLIENT_DATA

Triggered when new data is received from the server during Layer 7 classification of the connection.

Note: For UDP (and only UDP), the CLIENT_DATA event is automatically triggered for each UDP packet received.

Example

```
when CLIENT_DATA {
     if { [UDP::payload 50] contains "XYZ" } {
          matchaflex
          return
     }
}
```

Example

If a DNS request contains "abc", and if the request contains "xyz", match the rule.

```
when CLIENT_DATA {
     if { [UDP::payload 12 12] contains "abc" } {
          matchaflex
          return
     } elseif { [UDP::payload 12 12] contains "xyz" } {
          matchaflex
          return
     }
}
```

SERVER_DATA

Triggered when new data is received from the server during Layer 7 classification of the connection.

Operators

The following subsections describe the FleX operators.

Relational Operators

contains

Tests whether one string (string1) contains another string (string2).

Syntax

```
<string1> contains <string2>
```

Example

```
when SERVER_DATA {when CLIENT_DATA {
     if { [TCP::payload] contains "aol.com" } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

ends_with

Tests whether one string (string1) ends with another string (string2).

Syntax

```
<string1> ends_with <string2>
```

Example

```
when SERVER_DATA {
     if { [TCP::payload] ends_with "</html>" } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

equals

Tests whether one string equals another string.

Syntax

```
<string1> equals <string2>
```

Note: The “matches” operator uses the same comparison as the Tcl “string match” command, which functions like a cut-down regular expression. For the two strings to match, their contents must be identical except that the following special sequences may appear in the pattern:

- * – Matches any sequence of characters in string, including a null string.
- ? – Matches any single character in string.
- [chars] – Matches any character in the set given by chars. If a sequence of the form x-y appears in chars, then any character between x and y, inclusive, will match. When used with -nocase, the end points of the range are converted to lower case first. Whereas {[A-z]} matches '_' when matching case-sensitively ('_' falls between the 'Z' and 'a'), with -nocase this is considered to be like {[A-Za-z]}.
- \x – Matches the single character x. This provides a way of avoiding the special interpretation of the characters *?[]\ in a pattern.

Example

```
when CLIENT_DATA {
     if { [UDP::payload] matches {*\\aol\\[a-z].html} } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

matches

Tests whether one string matches another string.

Syntax

```
<string1> matches <string2>
```

Note: The "matches" operator uses the same comparison as the Tcl "string match" command, which functions like a cut-down regular expression.

For the two strings to match, their contents must be identical except that the following special sequences may appear in the pattern:

- * – Matches any sequence of characters in string, including a null string.
- ? – Matches any single character in string.
- [chars] – Matches any character in the set given by chars. If a sequence of the form x-y appears in chars, then any character between x and y, inclusive, will match. When used with -nocase, the end points of the range are converted to lower case first. Whereas {[A-z]} matches '_' when matching case-sensitively ('_' falls between the 'Z' and 'a'), with -nocase this is considered to be like {[A-Za-z]}. (This is probably what was meant in the first place).
- \x – Matches the single character x. This provides a way of avoiding the special interpretation of the characters *?[]\ in a pattern.

Valid Events

All events are supported.

matches_regex

Tests whether one string matches a regular expression.

Syntax

```
<string1> matches_regex <regex>
<string1> matches_regex <string2>
```

Tests if string2 is contained within string1.

Example

```
when CLIENT_DATA {
     if { [TCP::payload] matches_regex ".*www\.([\w]*)\.com*" } {
          matchaflex
          return
     } elseif { [TCP::payload] matches_regex ".*www\.([\w]*)\.edu*" } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

starts_with

Tests whether one string (string1) starts with another string (string2).

Syntax

```
<string1> starts_with <string2>
```

Example

```
when CLIENT_DATA {
     if { [UDP::payload] starts_with "OK" } {
          matchaflex
          return
     } elseif { [UDP::payload] starts_with "ERROR" } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

switch

Built-in TCL command. Evaluates one of several scripts, depending on a given value.

Syntax

```
switch ?options? string {pattern body ?pattern
body ...?}
```

Matches its string argument against each of the pattern arguments in order. As soon as it finds a pattern that matches string, it evaluates the following body argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last pattern argument is "default", then it matches anything. If no pattern argument matches string and no default is given, then the command returns an empty string.

If the initial arguments start with "-", then they are treated as options. The following options are currently supported:

- **-exact** – Use exact matching when comparing string to a pattern. This is the default.
- **-glob** – When matching string to the patterns, use glob-style matching (the same as implemented by the string match command).
- **-regexp** – When matching string to the patterns, use regular expression matching (the same as implemented by the regexp command).
- **--** – Marks the end of options. The argument following this one will be treated as string even if it starts with a "-".

Two syntaxes are provided for the pattern and body arguments.

The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands.

The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the pattern arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

If a body is specified as "-" it means that the body for the next pattern should also be used as the body for this pattern (if the next pattern also has a body of "-" then the body after that is used, and so on). This feature makes it possible to share a single body among several patterns.

Example

This example will return 2:

```
switch abc a - b {format 1} abc {format 2} default {format 3}
```

This example will return 3:

```
switch xyz {
     a
          -
     b
          {format 1}
     a*
          {format 2}
     default
          {format 3}
}
```

This example will inspect the TCP payload at offset 5, length 15 for string “www.domain.com” or “www.domain2.com”. If A string matches, the flow will be classified.

```
switch [string tolower [TCP::payload 5 15]] {
     www.domain.com { matchaflex }
     www.domain2.com {
          matchaflex
     }
     default { return }
}
```

Valid Events

All events are supported.

Logical Operators

and

Performs a logical “and” comparison between two values.

Syntax

```
<value1> and <value2>
```

Example

```
when CLIENT_DATA {
     if { ([TCP::data] starts_with "GET /abc") and ([TCP::payload] ends_with
"XXX") } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

not

Performs a logical "not" on a value.

Syntax

```
not <value>
```

Example

```
when CLIENT_DATA {
     if { not ([TCP::payload] starts_with "GET /") } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

or

Performs a logical "or" comparison between two values.

Syntax

```
<value1> or <value2>
```

Example

```
when CLIENT_DATA {
     if { ([TCP::payload] starts_with "GET /abc") or ([TCP::payload]
starts_with "GET /cde") } {
          matchaflex
     }
}
```

Valid Events

All events are supported.

Commands

The following subsections describe the aFleX commands.

GLOBAL Commands

b64decode

Returns the specified string, decoded from base-64. Returns NULL if there is an error.

Syntax

```
b64decode <string>
```

Example

```
when CLIENT_DATA {
     set encrypted [TCP::payload 0 8]
     set decrypted [b64decode $encrypted]
}
```

Valid Events

All events are supported.

b64encode

Returns the specified string, encoded as base-64. Returns NULL if there is an error.

Syntax

```
b64encode <string>
```

Example

```
when CLIENT_DATA {
     set encrypted [TCP::payload 0 8]
     set decrypted [b64decode $encrypted]
}
```

Valid Events

All events are supported.

clientside

Causes the specified aFleX commands to be evaluated under the client-side context. This command has no effect if the aFleX command is already being evaluated under the client-side context.

Syntax

```
clientside {<aFleX commands>}
```

Example

```
when SERVER_DATA {
     if { [IP::addr [clientside {IP::remote_addr}] equals 10.1.1.80] } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

domain

Parses the specified string as a dotted domain name and returns the last <count> portions of the domain name.

Syntax

```
domain <string> <count>
```

Valid Events

All events are supported.

event

Discontinue evaluating the specified aFleX event, or all aFleX events, on this connection. However, the aFleX script continues to run.

Syntax

```
event [<name>] [enable | disable] | [enable all |
disable all]
```

Valid Events

All events are supported.

findstr

Find a string within another string and return the string starting at the offset specified from the match.

Syntax

```
findstr <string> <search_string> [<skip_count>
[<terminator>]
```

Finds the string <search_string> within <string> and returns a sub-string based on the <skip_count> and <terminator> from the matched location. Note the following:

- The <terminator> argument may be either a character or length.
- If the <skip_count> argument is not specified, it defaults to zero.
- If the <terminator> argument is not specified, it defaults to the end of the string.
- This command, without <skip_count> or <terminator>, is equivalent to the following Tcl command:

```
string range <string> [string first <string>
<search_string>] end
```

Example

```
when CLIENT_DATA {
     if { [findstr [TCP::payload] "type=" 5 "&"] eq "cgi" } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

getfield

Splits a string on a character or string, and returns the string corresponding to the specific field.

Syntax

```
getfield <string> <split> <field_number>
```

Example

```
when CLIENT_DATA {
     [getfield “www.a10networks.com” ":" 1]
}
when CLIENT_DATA {
     if { [TCP::payload] contains "domain.com"} {
          [getfield “xxx.domain.com” ".domain.com" 1]
     }
}
```

Valid Events

All events are supported.

htonl

Convert the unsigned integer from host byte order to network byte order.

Syntax

```
htonl <hostlong>
```

Example

```
when CLIENT_DATA {
     set hostlong 12345678
     set netlong [htonl $hostlong]
}
```

Valid Events

All events are supported.

htons

Convert the unsigned short integer from host byte order to network byte order.

Syntax

```
htons <hostshort>
```

Example

```
when CLIENT_DATA {
     set hostshort 1234
     set netshort [htons $hostshort]
}
```

Valid Events

All events are supported.

if

Ask a true or false question and, depending on the answer, takes some action.

Note: The maximum number of `if` statements that you can nest in an aFleX policy is 100.

Syntax

```
if { <expression> } {<statement_command>}
elseif { <expression> } {<statement_command>}
```

matchaflex

Match an aFleX rule for Layer 7 classification.

Syntax

```
matchaflex
```

Example

```
when SERVER_DATA {
    if { [L7CL::pkts rev] == 1 } {
        if { ([TCP::payload] starts_with "HTTP") } {
            matchaflex
            return
        }
    }
}
```

Valid Events

All events are supported.

ntohl

Convert the unsigned integer from network byte order to host byte order.

Syntax

```
ntohl <netlong>
```

Example

```
when CLIENT_DATA{
     set netlong 12345678
     set hostlong [ntohl $netlong]
}
```

Valid Events

All events are supported.

ntohs

Convert the unsigned short integer from network byte order to host byte order.

Syntax

```
ntohs <netshort>
```

Example

```
when CLIENT_DATA {
set netshort 1234
set hostshort [ntohs $netshort]
}
```

Valid Events

All events are supported.

return

Terminates the aFleX script and returns execution to internal packet processing functions from the aFleX event callback.

serverside

Causes the specified aFleX command or commands to be evaluated under the server-side context. This command has no effect if the aFleX policy is already being evaluated under the server-side context.

Syntax

```
serverside { <aFleX command> }
```

Example

```
when CLIENT_DATA {
     if {[IP::addr [serverside {IP::remote_addr}] equals 10.1.1.80] } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

substr

Returns a sub-string named <string>, based on the values of the <skip_count> and <terminator> arguments.

Syntax

```
substr <string> <skip_count> [<terminator>]
```

Notes:

- The <skip_count> and <terminator> arguments are used in the same way as they are for the findstr command.
- The <skip_count> argument is the index into <string> of the first character to be returned, where 0 indicates the first character of <string>.
- The <terminator> argument can be either the substring length or the substring terminating string.
- If <terminator> is an integer, the returned string will include that many characters, or up to the end of the string, whichever is shorter.
- If <terminator> is a string, the returned string will include characters up to but not including the first occurrence of the string.
- If <terminator> is a string which does not occur in the search space, from <skip_count> to the end of <string> is returned.
- This command is equivalent to the Tcl string range command except that the value of the <terminator> argument may be either a character or a count.

Example

```
when CLIENT_DATA {
     set uri [substr $uri 1 "?"]
}
log "[substr "abcdefghijklm" 2 "x"]"
log "[substr "abcdefghijklm" 2 "gh"]"
log "[substr "abcdefghijklm" 2 4]"
log "[substr "abcdefghijklm" 2 20]"
log "[substr "abcdefghijklm" 2 0]"
The above example logs the following:
cdefghijklm
cdef
cdef
cdefghijklm
cdefghijklm
```

Valid Events

All events are supported.

when

Specify an event in an aFleX script. All aFleX events begin with a `when` command. You can specify multiple `when` commands within a single aFleX script.

Syntax

```
when <event_name>
```

Example

```
when CLIENT_DATA {
     if { [IP::addr [IP::client_addr] equals 10.10.10.10] } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

IP Commands

IP::addr

Performs comparison of IP address/subnet/supernet to IP address/subnet/supernet. Returns 0 if no match, 1 for a match.

Note: This command does ***not*** perform a string comparison. To perform a literal string comparison, simply compare the 2 strings with the appropriate operator (equals, contains, starts_with, and so on) rather than using the IP::addr comparison.

Syntax

```
IP::addr <addr1>[/<mask>] equals <addr2>[/<mask>]
  IP::addr
```

Example

To perform comparison of IP address 10.10.10.1 with subnet 10.0.0.0/8. (Will return 1, since it is a match.)

```
[IP::addr 10.10.10.1 equals 10.0.0.0/8]
```

To perform comparison of client-side IP address with subnet 10.0.0.0/8. (Will return 1 or 0, depending on client IP address.)

```
[IP::addr [IP::client_addr] equals 10.0.0.0/8]
```

To match traffic for a specific client IP address.

```
when CLIENT_DATA {
     if { [IP::addr [IP::client_addr] equals 10.10.10.10] } {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

IP::client_addr

Returns the client IP address of a connection. This command is equivalent to the command `clientside { IP::remote_addr }`.

Syntax

```
IP::client_addr
```

Example

```
when CLIENT_DATA {
     if { [IP::addr [IP::client_addr] equals 10.10.10.10] } {
          matchaflex
          return
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

IP::local_addr

This command is primarily useful for generic rules that are re-used. Also, it is useful in reusing the connected endpoint in another statement or to make routing type decisions. You can also specify the `IP::client_addr` and `IP::server_addr` commands.

Syntax

`IP::local_addr`

Returns the IP address of the EX being used in the connection. In the client-side context, this is the destination IP address (virtual IP address). In the serverside context, this is the source IP address (SNAT address if SNAT is used, else spoofed client IP address).

Example

```
when CLIENT_DATA {
     if { [IP::addr [IP::local_addr] equals 172.16.32.2] } {
          matchaflex
          return
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

IP::protocol

Returns the IP protocol value.

Syntax

`IP::protocol`

Example

```
when CLIENT_DATA {
     if { [IP::protocol] == 6 } {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

IP::remote_addr

Returns the IP address of the host on the far end of the connection. In the clientside context, this is the client IP address. In the serverside context this is the node IP address. You can also specify the `IP::client_addr` and `IP::server_addr` commands, respectively.

Syntax

```
IP::remote_addr
```

Example

```
when CLIENT_DATA {
     if { [IP::addr [IP::remote_addr] equals 206.0.0.0/255.0.0.0] } {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

IP::server_addr

Returns the server's (node's) IP address, once a serverside connection has been established. This command is equivalent to the command serverside {IP::remote_addr}. The command returns 0 if the serverside connection has not been made.

Syntax

```
IP::server_addr
```

Example

```
when CLIENT_DATA {
     if { [IP::addr [IP::server_addr] equals 192.168.3.39] } {
          matchaflex
          return
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

IP::stats

Supplies information about the number of packets or bytes being sent or received in a given connection.

Syntax

```
IP::stats pkts in
IP::stats pkts out
IP::stats pkts
IP::stats bytes in
IP::stats bytes out
IP::stats bytes
IP::stats age
```

`IP::stats pkts in`

Returns number of packets received

`IP::stats pkts out`

Returns number of packets sent

`IP::stats pkts`

Returns a Tcl list of packets in and packets out

`IP::stats bytes in`

Returns number of bytes received

`IP::stats bytes out`

Returns number of bytes sent

`IP::stats bytes`

Returns Tcl list of bytes in and bytes out

IP::tos

Selects a different pool of servers based on the ToS level within a packet. The Type of Service (ToS) standard is a means by which network equipment can identify and treat traffic differently based on an identifier. As traffic enters the site, the EX device can apply a rule that sends the traffic to different pools of servers based on the ToS level within a packet.

Note: This command replaces the `ip_tos` command.

Syntax

```
IP::tos
```

Selects a different pool of servers based on the ToS level within a packet.

Example

```
when CLIENT_DATA {
     if { [IP::tos] == 16 } {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

IP::ttl

Returns the time to live (TTL) of the current packet being acted upon.

Syntax

```
IP::ttl
```

Example

```
when CLIENT_DATA {
     if { [IP::ttl] < 3 } {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

IP::version

Returns the version of the current packet being acted upon.

Syntax

```
IP::version
```

Example

```
when CLIENT_DATA {
     if {[IP::version] eq 6} {
          matchaflex
          return
     }
}
```

Valid Events

```
CLIENT_DATA
```

Layer 7 Classification Commands

L7CL::get_flag

Returns the Layer 7 classification flags, by offset.

Syntax

```
L7CL::get_flag <offset>
```

Example

```
when SERVER_DATA {
    if { [L7CL::pkts rev] == 1 } {
        if { ([L7CL::get_flag 1] == 1) and ([TCP::payload] starts_with "HTTP")
} {
            matchaflex
            return
        }
    }
}
when CLIENT_DATA {
    if { [L7CL::pkts fwd] == 1 } {
        if { [TCP::payload] contains "Host: 192.168.3.136" } {
            L7CL::set_flag 1
            return
        }
    }
}
```

Valid Events

All events are supported.

L7CL::pkt_cnt

Returns the count of packets that have been Layer-7 classified. The count includes the current packet.

Syntax

```
L7CL::pkt_cnt [<fwd> | <rev>]
```

Example

```
when SERVER_DATA {
    if { [L7CL::pkts rev] == 1 } {
        if { ([TCP::payload] starts_with "HTTP") } {
            matchaflex
            return
        }
    }
}
when CLIENT_DATA {
    if { [L7CL::pkts fwd] == 1 } {
        if { [TCP::payload] contains "Host: 192.168.3.136" } {
            matchaflex
            return
        }
    }
}
```

Valid Events

All events are supported.

L7CL::set_flag

Sets the Layer 7 classification flags, by offset.

Syntax

```
L7CL::set_flag <offset>
```

Example

See L7CL::get_flag.

Valid Events

All events are supported.

TCP Commands

TCP::client_port

Returns the TCP port/service number of the specified client. This command is equivalent to the command `clientside { TCP::remote_port }` and to `client_port`.

Syntax

```
TCP::client_port
```

the variable `local_port`.

Example

```
when CLIENT_DATA {
     if { [TCP::client_port] > 1000 } {
          matchaflex
          return
     }
}
```

Valid Events

All events are supported.

TCP::local_port

Returns the local TCP port/service number. This command is equivalent to the variable `local_port`.

Example

```
when CLIENT_DATA {
     if {[IP::protocol] == 47 || [TCP::local_port] == 1723} {
         # GRE used by MS PPTP server, TCP control channel
         matchaflex
         return
     } elseif {[IP::protocol] == 50 || [IP::protocol] == 51 ||
[UDP::local_port] == 500} {
         # AH and ESP used by IPSec, IKE used by IPSec
         matchaflex
         return
     } elseif {[IP::protocol] == 115} {
         # L2TP Protocol server
         matchaflex
         return
     }
}
```

Syntax

```
TCP::local_port
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

TCP::mss

Returns the on-wire Maximum Segment Size (MSS) for a TCP connection.

Syntax

```
TCP::mss
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

TCP::payload

Returns the accumulated TCP data content, or replaces collected payload with the specified data.

Syntax

```
TCP::payload [<size>]
TCP::payload <offset> <size>
TCP::payload length
TCP::payload replace <offset> <size> <data>
TCP::payload [<size>]
```

Returns the accumulated TCP data content.

```
TCP::payload <offset> <size>
```

Returns the accumulated TCP data content start from <offset>.

```
TCP::payload replace <offset> <size> <data>
```

Replaces collected payload with the given data.

```
TCP::payload length
```

Returns the amount of accumulated TCP data content in bytes.

Example

```
when CLIENT_DATA {
     if { [TCP::payload] contains "flower" } {
          matchaflex
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

TCP::remote_port

Returns the remote TCP port/service number.

When used with the `clientside` command (that is, `clientside TCP::remote_port`), the `TCP::remote_port` command is equivalent to the `TCP::client_port` command.

When used with the `serverside` command (that is, `serverside TCP::remote_port`), the `TCP::remote_port` command is equivalent to the `TCP::server_port` command.

Syntax

```
TCP::remote_port
```

Example

```
when SERVER_DATA {
     if { [TCP::remote_port] == 8080 } {
          matchaflex
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

TCP::server_port

Returns the TCP port/service number of the specified server. This command is equivalent to the command `serverside { TCP::remote_port }` and to the BIG-IP 4.x variable server_port.

Syntax

`TCP::server_port`

Example

```
when SERVER_DATA {
     if { [TCP::server_port] > 1000 } {
          matchaflex
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

UDP Commands

UDP::client_port

Returns the UDP port/service number of the client system. This command is equivalent to the command `clientside { UDP::remote_port }`.

Syntax

`UDP::client_port`

Example

```
when CLIENT_DATA {
     if { [UDP::client_port] equals 80 } {
          matchaflex
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

UDP::local_port

Returns the local UDP port/service number.

Syntax

```
UDP::local_port
```

Example

```
when CLIENT_DATA {
     if {[IP::protocol] == 47 || [TCP::local_port] == 1723} {
          # GRE used by MS PPTP server, TCP control channel
          matchaflex
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

UDP::mss

Returns the on-wire Maximum Segment Size (MSS) for a UDP connection.

Syntax

```
UDP::mss
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

UDP::payload

Returns the content or length of the current UDP payload.

Syntax

```
UDP::payload [<size>]
```

Returns the content of the current UDP payload.

```
UDP::payload length
```

Returns the length, in bytes, of the current UDP payload.

```
UDP::payload offset size
```

Returns the content of the current UDP payload from <offset>.

Example

```
when CLIENT_DATA {
     if { [UDP::payload 12 20] contains "a10networks" } {
          matchaflex
     }
}
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

UDP::remote_port

Returns the remote UDP port/service number.

Syntax

```
UDP::remote_port
```

Valid Events

```
CLIENT_DATA, SERVER_DATA
```

UDP::server_port

Returns the UDP port/service number of the server. This command is equivalent to the command `serverside { UDP::remote_port }`.

Syntax

```
UDP::server_port
```

Example

```
when SERVER_DATA {
     if { [UDP::server_port] equals 80 } {
          matchaflex
     }
}
```

Valid Events

`CLIENT_DATA, SERVER_DATA`

Corporate Headquarters

A10 Networks, Inc.
2309 Bering Dr.
San Jose, CA 95131-1125 USA

Tel: +1-408-325-8668 (main)
Tel: +1-408-325-8676 (support - worldwide)
Tel: +1-888-822-7210 (support - toll-free in USA)
Fax: +1-408-325-8666

www.a10networks.com