

Open Source Detectors Developers Guide

Revision 3.0
May 29, 2014

Copyright © 2014 Cisco and/or its affiliates. All rights reserved.

Table of Contents

1 Overview	4
2 Detector Code Structure	4
2.1 Detector Info Header	5
2.2 Include Required Libraries	5
2.3 DetectorPackageInfo	6
2.4 Detector Initialize	6
2.5 Detector Validate	7
2.6 DetectorFini	7
3 Debugging	7
4 Lua-C API	7
4.1 Detector Api	7
4.2 Detector Flow API	15
5 Appendix	16
5.1 Flow Flags	16

1 Overview

In Snort version 2.9.7, Cisco will release a new dynamic preprocessor OpenAppID, which will add application identification to Snort capabilities. Application identification can be used to view how applications are using network resources and to enforce application aware rules to control and manage applications running on network.

Cisco will release open source code for hundreds of application detectors that can be used to identify frequently used applications. Users are free to copy and modify Cisco-provided detectors and create new detectors. The detectors will be small Lua programs that use a C-Lua API to interact with OpenAppID preprocessor. This document presents developer's guide to understand detector design and write a custom detector.

2 Detector Code Structure

A detector has the following main components. The Detector Info Header, the included required libraries, the DetectorPackageInfo, the DetectorInit function, the DetectorValidator function, and the DetectorClean. We will describe each of them in more details below.

```

--[[
detection_name: CraftBeersFests
version: 2
description: CraftBeersFests.
--]]

require "DetectorCommon"
local DC = DetectorCommon

DetectorPackageInfo = {
  name = 'craftbeersfests',
  proto = DC.ipproto.tcp,
  client = {
    init = 'DetectorInit',
    validate = DetectorValidate,
    clean = DetectorClean,
    minimum_matches = 0
  },
  server = {
    init = nil,
    validate = nil,
    clean = nil
  },
}

function DetectorInit(detectorInstance)

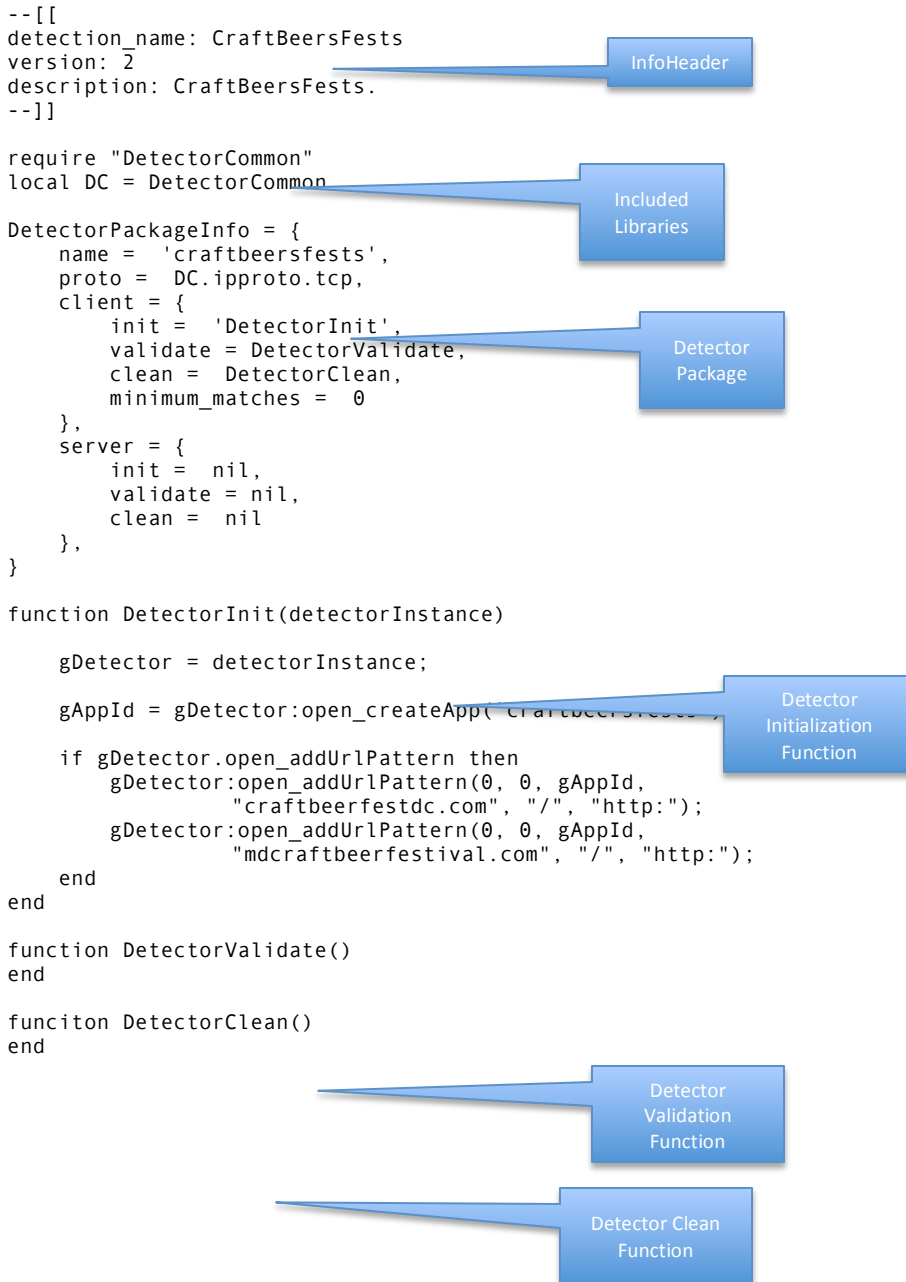
  gDetector = detectorInstance;

  gAppId = gDetector:open_createApp('craftbeersfests',
  if gDetector.open_addUrlPattern then
    gDetector:open_addUrlPattern(0, 0, gAppId,
      "craftbeerfestdc.com", "/", "http:");
    gDetector:open_addUrlPattern(0, 0, gAppId,
      "mdcraftbeerfestival.com", "/", "http:");
  end
end

function DetectorValidate()
end

function DetectorClean()
end

```



2.1 Detector Info Header

This is an optional comment block that describes the detector. Cisco detectors will contain information in the following format. User detectors can continue to use the same format, add more fields or completely skip the info header. This has no impact on the functionality of the detector.

2.2 Include Required Libraries

In order to keep the detectors short, some commonly used code can be placed into a library. Here the Cisco detector is including a library DetectorCommon.lua and creating a shortcut 'DC' to it.

```
require "DetectorCommon"  
local DC = DetectorCommon
```

Cisco libraries must be placed into the `odp/libs` subdirectory under the directory where Cisco Open Detector Package (ODP) was installed. This path is automatically included in the Lua path. Users can create more libraries and place them in `<ODP_install_dir>/custom/libs`, which is also automatically included in the Lua path.

2.3 DetectorPackageInfo

The `DetectorPackageInfo` structure is required in each detector. It identifies client and server functions that will be called to initialize, validate (process packets) and cleanup the detector. `OpenAppID` preprocessor reads this structure after loading Lua code and calls initialization functions.

The structure has the following elements:

1. `DetectorPackageInfo.name`
This is a name for the detector that is used for logging purpose.
2. `DetectorPackageInfo.proto`
Protocol value. It can be `DC.ipproto.tcp` or `DC.ipproto.udp`.
3. `DetectorPackageInfo.client`
If the detector identifies client side application (for example Firefox) then this structure is populated. Detectors for payload application (example Facebook) will provide client section only. The following functions can be provided:
 - `init`
Name of callback function that initialize a detector. See “Detector Initialize” section for details.
 - `validate`
Name of callback function that process packets in the detector. The function typically inspects packet contents and may use stored results from previous packets to detect an application. Before finishing, the functions call an appropriate API function to indicate results of detection. These functions are not required for some specific applications. See “Detector Validate” for more details.
 - `clean`
Name of callback function that perform cleanup when Snort is exiting. The function is optional and may be omitted in the `DetectorPackageInfo` structure.
4. `DetectorPackageInfo.server`
If the detector identifies a server side application (for example Apache web server) then this structure is populated. The structure provides `init`, `validate` and `clean` function names that have same purpose as in client side.

2.4 Detector Initialize

Each detector must have an initializer function that is present in the `DetectorPackageInfo` structure. `OpenAppID` preprocessor will call this function directly upon loading the detector.

The function is given `detectorInstance`, an instance of `Detector` class, which should be stored globally and used for calling all Lua-C API functions. The function may perform one or more of the following:

1. Create a new application name by calling `open_createApp()`.
2. Setup fast patterns and the port for an application. These are used for selecting a detector for a flow. See `service_registerPattern()` and `service_addPorts()`.
3. Add patterns for specific headers for HTTP . See `open_addUrlPattern()` etc.

2.5 Detector Validate

The validation function, provided in the `DetectorPackageInfo` structure, is called when the `OpenAppID` preprocessor determines the detector is viable for deeper inspection to detect an application. It performs the same logic steps as a detector written in C. It can be a state driven pattern match that spans multiple packets in a flow or a simple straight pattern match with any packet.

```
function DetectorValidator()
    local size = gDetector:getPacketSize()
    local dir = gDetector:getPacketDir()

    if (size == 0 or dir ~= 1) then
        gDetector:inProcessService()
        return DC.serviceStatus.inProcess
    end

    if ((size >= 35) and (gDetector:getPcreGroups("stream:stream"))) then
        gDetector:addService(gServiceId, "", "", 692)
        return DC.serviceStatus.success
    end

    gDetector:failService()
    return serviceFail(context)
end
```

Validate functions are not called if the actual application is HTTP, SSL, and SIP based. For these applications, Snort preprocessors parse protocol headers and make them available for pattern matching through C-Lua API functions `open_addUrlPattern()` is an example of one such function. `CraftBeersFests` Customer detector provided a validator function just to show program structure.

2.6 DetectorFini

This function is called when a detector is destroyed during Snort exit. Note that Lua performs garbage collection automatically when an object is not referenced anymore so this function does free memory. Possible uses of the function are to print statistics about flow, packets, application detected etc.

3 Debugging

See <http://lua-users.org/wiki/DebuggingLuaCode> for information on debugging Lua. One can also use print statements for debugging.

4 Lua-C API

The following sections provide detailed view into Lua-C API functions. The documentation sometimes refers to `DetectorCommon.lua` file, which is include in the ODP package and installed under `odp/libs` subdirectory. This is a common library file that is included in all detector Lua files. It provides common definitions and helper functions.

4.1 Detector Api

4.1.1 Client Side API

4.1.1.1 `int client_registerPattern (protocol, pattern, patternSize, position)`

Register pattern hints for this detector selection.

Parameters:

<i>Protocol</i>	IP Protocol. See <code>ipproto</code> values defined in <code>DetectorCommon.lua</code> . Example <code>DC.ipproto.tcp</code> . Values are same as protocol values in <code>/usr/include/netinet/in.h</code> or equivalent system header file.
<i>pattern</i>	ASCII or binary pattern
<i>patternSize</i>	Size of pattern in bytes
<i>position</i>	Position in packet payload where pattern should match. First byte is position 1. Position 0 means any position.

Returns:

`int` – 0 if successful and -1 otherwise.

4.1.2 Server Side API

4.1.2.1 `int service_registerPattern (protocol, pattern, patternSize, position)`

Register a pattern for fast pattern matching. Lua detector calls this function to register a pattern for fast pattern matching.

Parameters:

<i>protocol</i>	IP Protocol. See <code>ipproto</code> values defined in <code>DetectorCommon.lua</code> . Example <code>DC.ipproto.tcp</code>
<i>pattern</i>	ASCII or binary pattern
<i>patternSize</i>	Size of pattern in bytes
<i>position</i>	Position in packet payload where pattern should match. First byte is position 1. Position 0 means any position.

Returns:

`status/stack` - 0 if successful, -1 otherwise.

4.1.2.2 `int service_addPorts (protocol, port)`

Lua detectors call this function to register ports on which a given service is expected to run.

Parameters:

<i>Protocol</i>	IP Protocol. See <code>ipproto</code> values defined in <code>DetectorCommon.lua</code> . Example <code>DC.ipproto.tcp</code>
<i>Port</i>	Port number to register

Returns:

`int` - Number of elements on stack, which is always 1.
`status/stack` - 0 if successful, -1 otherwise.

4.1.2.3 `int service_removePorts ()`

Remove ports for this service when exiting.

Parameters:

Returns:

`status/stack` - 0 if successful, -1 otherwise.

4.1.2.4 `int service_addAppIdDataToFlow (servicePort)`

Add App ID related data to a future flow. Currently only service port number and validator function name are added to future flow. The data is used to confirm a future flow matches a pre-selected service. The validator function name is picked from C side so is not specified in API call.

Parameters:

<code>servicePort</code>	Service port number
--------------------------	---------------------

Returns:

0 if successful, -1 otherwise.

4.1.2.5 `int service_failService ()`

Function confirms the flow is not running this service.

Parameters:

Returns:

int - values from enum `serviceStatus` in `DetectorCommon.lua` file.

4.1.2.6 `int service_markIncompleteData ()`

Detector uses this function to indicate an error in application identification due to incomplete knowledge/data. The flow is not inspected anymore. Next related flow with the same responder IP, Port and Protocol, however, will be given to the same detector to allow it to hopefully gather a complete set of data to make a determination.

This function is used in cases where a precondition for detecting an application is violated therefore the detector cannot proceed with detection. As an example, an application may be detected by a pattern "ServicePattern" in the responder packet only when the initiator sends pattern "clientPattern". If "clientPattern" is not seen from the initiator, the detector will declare the flow incompatible, so that it still gets the next flow.

In contrast, `service_failService()` would cause the next flow to go to another detector.

Parameters:

Returns:

int - values from enum `serviceStatus` in `DetectorCommon.lua` file.

4.1.2.7 `int service_inProcessService ()`

Detector uses this function to indicate the detector needs more packets to determine the application. Subsequent packets are given to this detector for more analysis.

Parameters:

Returns:

int - values from enum `serviceStatus` in `DetectorCommon.lua` file.

4.1.3 *Common API*

4.1.3.1 `int getPacketSize ()`

Gets length of TCP/UDP payload of current packet.

Parameters:

Returns:

packetSize - length of TCP/UDP payload of current packet, if successful. NIL otherwise.

4.1.3.2 `int getPacketDir ()`

Gets packet direction. A flow/session maintains initiator and responder sides. A packet direction is determined in relation to the original initiator.

Parameters:

Returns:

packetDir - direction of packet on stack, if successful. NIL otherwise.

4.1.3.3 int matchSimplePattern (pattern, patternSize, position)

Performs a simple pattern comparison against packet payload.

Parameters:

<i>pattern</i>	ASCII or binary pattern
<i>patternSize</i>	Size of pattern in bytes
<i>position</i>	Position in packet payload where pattern should match. First byte is position 1. Position 0 means any position.

Returns:

memcmpResult - returns -1, 0, or 1 based on memcmp result.

4.1.3.4 int getPcreGroups (pattern, position)

Performs a PCRE (Perl Compatible Regular Expression) match with grouping. A simple regular expression match with no grouping can also be performed.

Parameters:

<i>pattern</i>	PCRE pattern
<i>position</i>	Position in packet payload where pattern should match. First byte is position 1. Position 0 means any position.

Returns:

matchedStrings - matched strings are pushed on stack starting with group 0. There may be 0 or more

4.1.3.5 int getL4Protocol ()

Gets protocol field value from IP header.

Parameters:

Returns:

IP protocol values if successful; for example TCP(6), UDP(17).
0 otherwise.

4.1.3.6 ip getPktSrcIPAddr ()

Gets source IPv4 address from IP header.

Parameters:

Returns:

IPv4 - Source IPv4 address.

4.1.3.7 ip getPktDstIPAddr ()

Gets destination IPv4 address from IP header.

Parameters:

Returns:

IPv4 - destination IPv4 address.

4.1.3.8 int getPktSrcPort ()

Gets source port number from IP header.

Parameters:

Returns:

portNumber - source port number.

4.1.3.9 int getPktDstPort ()

Gets destination port number from IP header.

Parameters:

Returns:

portNumber - destination Port number.

4.1.3.10 int getPktCount ()

Gets packet count. This is used mostly for printing a packet sequence number when testing with a pcap file.

Parameters:

Returns:

packetCount - Total packet processed by RNA.

4.1.3.11 void getFlow ()

Gets flow object from a detector object. The flow object is then used with flowApi. A new copy of flow object is provided with every call.

Parameters:

Returns:

4.1.3.12 void logMessage (level, message)

Logs messages from detectors into wherever /etc/syslog.conf directs them. An example is: detector:log(DC.logLevel.warning, 'a warning')

Parameters:

<i>level</i>	Level of message. See DetectorCommon.lua for enumeration.
<i>message</i>	Message to be logged.

4.1.3.13 Void addContentTypePattern (pattern, appId)

Adds pattern for content type HTTP header.

Parameters:

<i>pattern</i>	Pattern to match content type HTTP header.
<i>appId</i>	Application identifier.

Returns:

4.1.3.14 void addSipUserAgent (appId, version, pattern)

Adds pattern to detect SIP client.

Parameters:

<i>appId</i>	AppId assigned to SIP client.
<i>Version</i>	Version of SIP client. Not used in open source.
<i>pattern</i>	Pattern to match on SIP user agent header.

Returns:

4.1.3.15 `addSipServer (appId, version, pattern)`

Adds pattern to detect SIP server.

Parameters:

<i>appId</i>	AppId assigned to SIP server.
<i>Version</i>	Version of SIP server. Not used in open source.
<i>pattern</i>	Pattern to match on SIP server header.

Returns:

4.1.3.16 `void addHostPortApp (type, appId, ip, port, protocol)`

Adds IP address, port, and protocol to identify future flows as a specific application.

Parameters:

<i>type</i>	Always 0.
<i>AppId</i>	Application identifier
<i>IP</i>	IPv6 or IPv4 pattern in ASCII string format. Subnet format not supported.
<i>Port</i>	Port number
<i>Protocol</i>	TCP or UDP

Returns:

void

4.1.3.17 `int registerAppId (AppId)`

Adds server-side application id to a Snort session.

Parameters:

<i>AppId</i>	Application Identifier
--------------	------------------------

Returns:

0 is successful, -1 otherwise.

4.1.3.18 `void addAppUrl (serviceAppId, clientAppId, clientAppType, payloadAppId, payloadAppType, hostPattern, pathPattern, schemePattern, queryPattern, appId)`

This function will add the specified URL pattern to the pattern list to be searched against when HTTP traffic is detected. This can be used to further identify the application type for this traffic. If a match is detected, the traffic will be further tagged with the specified application identification data.

Parameters:

<i>serviceAppId</i>	Service application ID (such as HTTP)
<i>clientAppId</i>	Client application ID (such as a browser)
<i>clientAppType</i>	Client application ID type (legacy – currently not used)
<i>payloadAppId</i>	Payload application ID (legacy)
<i>payloadAppType</i>	Payload application ID type (legacy – currently not used)
<i>hostPattern</i>	Pattern to match for host in URL; for example “examplewebsite.com”
<i>pathPattern</i>	Pattern to match for path in URL; for example “/example/path”
<i>schemePattern</i>	Pattern to match for scheme; for example “http:”
<i>queryPattern</i>	Pattern to match for query in URL; for example “examplequery=somevalue”
<i>appId</i>	Application ID (such as a user-defined application)

Returns: void

4.1.3.19 void addRTMPUrl (serviceAppId, clientAppId, clientAppType, payloadAppId, payloadAppType, hostPattern, pathPattern, schemePattern, queryPattern, appId)

This function will add the specified URL pattern to the pattern list to be searched against when an RTMP session is detected. This can be used to further identify the application type for this traffic. If a match is detected, the traffic will be further tagged with the specified application identification data.

Parameters:

<i>serviceAppId</i>	Service application ID (such as HTTP)
<i>clientAppId</i>	Client application ID (such as a browser)
<i>clientAppType</i>	Client application ID type (legacy – currently not used)
<i>payloadAppId</i>	Payload application ID (legacy)
<i>payloadAppType</i>	Payload application ID type (legacy – currently not used)
<i>hostPattern</i>	Pattern to match for host in URL; for example “examplewebsite.com”
<i>pathPattern</i>	Pattern to match for path in URL; for example “/example/path”
<i>schemePattern</i>	Pattern to match for scheme; for example “http:”
<i>queryPattern</i>	Pattern to match for query in URL; for example “examplequery=somevalue”
<i>appId</i>	Application ID (such as a user-defined application)

Returns: void

4.1.3.20 void addSSLCertPattern (type, appId, pattern)

This function will add the specified pattern to the pattern list of host names to be searched when SSL traffic is detected. This information is retrieved from the client’s certificate request to the server. If the pattern matches, the specified application ID will be tagged to the traffic.

Parameters:

<i>Type</i>	Type of ID: web application (0), or client (1). If the type is web application, an SSL client will be assumed, and the traffic payload type will be tagged with the specified application. If the type is client, then the specified application ID will be associated with the client ID of the traffic.
<i>appId</i>	Application ID
<i>Pattern</i>	Pattern to match against SSL host name; for example “examplewebsite.com”

Returns: void

4.1.3.21 void addSSLNamePatter (type, appId, pattern)

This function will add the specified pattern to the pattern list of SSL common/organization names to be searched when SSL traffic is detected. This information is retrieved from the SSL certificate exchanged from the server (common name and organization name). If the pattern matches, the specified application ID will be tagged to the traffic.

Parameters:

<i>type</i>	Type of ID: web application (0), or client (1). If the type is web application, an SSL client will be assumed, and the traffic payload type will be tagged with the specified application. If the type is client, then the specified application ID will be associated with the client ID of the traffic.
<i>appId</i>	Application ID
<i>pattern</i>	Pattern to match against SSL host name; for example “Example Common Name”

Returns: void

4.1.4 Open API

This API provides simplified functions to support open source detectors written by users.

4.1.4.1 int open_createApp (appName)

Converts appName (a string) to an AppId (unique number). If appName does not match any existing application name then a new application is created and a unique AppId is assigned dynamically. For existing applications, the matching AppId is returned. This AppId should be used subsequently with C-Lua API wherever an AppId is required. Dynamic AppId values can change between different Snort runs.

Parameters:

<i>appName</i>	A unique name for user created application.
----------------	---

Returns:

A valid appId or NIL.

4.1.4.2 int open_addClientApp (clientAppId, serviceAppId)

Add client-side application. ServiceAppId is for services that can be inferred from the client-side application.

Parameters:

<i>clientAppId</i>	Client-side AppId
<i>serviceAppId</i>	Service AppId inferred from client AppId.

Returns:

0 if successful, -1 otherwise.

4.1.4.3 int open_addServiceApp (serviceAppId)

Add server-side application.

Parameters:

<i>serviceAppId</i>	Service AppId
---------------------	---------------

Returns:

0 if successful, -1 otherwise.

4.1.4.4 int open_addPayloadApp (payloadAppId)

Add server-side application.

Parameters:

<i>payloadAppId</i>	Payload AppId
---------------------	---------------

Returns:

0 if successful, -1 otherwise.

4.1.4.5 void open_addHttpPattern (patternType, Sequence, serviceAppId, clientAppId, payloadAppId, pattern)

API to add patterns for user agent and other special purpose patterns. Not to be used by new detectors.

Parameters:

<i>patternType</i>	Determines location where pattern should appear. Valid values are host (1), user agent (2), URL (3)
<i>Sequence</i>	0 single, 5 user agent header.
<i>serviceId</i>	Service AppId

<i>ClientAppId</i>	Client AppId
<i>PayloadAppId</i>	Payload AppId
<i>Pattern</i>	Pattern to match

Returns: void

4.1.4.6 void open_addUrlPattern (serviceAppId, clientAppId, payloadAppId, hostPattern, pathPattern, schemePattern)

API to add patterns for user agent and other special purpose patterns. Not to be used by new detectors.

Parameters:

<i>serviceld</i>	Serviceld
<i>ClientAppId</i>	ClientAppId
<i>PayloadAppId</i>	PayloadAppId
<i>hostPattern</i>	Pattern to match host HTTP header
<i>pathPattern</i>	Pattern to match in path of URL
<i>SchemePattern</i>	Pattern to match in scheme; for example "http"

Returns: void

4.2 Detector Flow API

4.2.1 Detector Flow API

4.2.1.1 void clearFlowFlag (flowFlags)

Clears the specified flow-related flag(s). See flowFlags in DetectorCommon.lua for a list of flow flags. Refer to the appendix for a list of available flags.

Parameters:

<i>flowFlags</i>	Flow flags to be reset to 0.
------------------	------------------------------

Returns:

Void

4.2.1.2 unsigned int getFlowFlag (flowFlag)

Gets the value of the specified flow-related flag(s). See flowFlags in DetectorCommon.lua for a list of flow flags. Refer to the appendix for a list of available flags.

Parameters:

<i>flowFlag</i>	Flow flag value to get.
-----------------	-------------------------

Returns:

int - value of a given flag.

4.2.1.3 void setFlowFlag (flowFlags)

Sets the specified flow-related flag(s). See flowFlags in DetectorCommon.lua for list of flow flags. Refer to the appendix for a list of available flags.

Parameters:

<i>flowFlags</i>	Flow flags to be set (value 1)
------------------	--------------------------------

Returns:

void

4.2.1.4 getFlowKey (void)

Gets a unique flow/session key. This key can be uniquely associated with this specific flow, so it can be used to maintain flow-specific data for the lifetime of the flow. Maintaining flow-specific data on the Lua side can be

very expensive so this should be used only when needed.

Parameters:

Returns:

flowKey- A 4 byte flow key

4.2.1.5 void* createFlow (srcAddr, srcPort, dstAddr, dstPort, protocol)

Creates a new flow and returns user data for newly created flow. The new flow creates an expected channel in Snort.

Parameters:

<i>srcAddress</i>	Source address of the flow
<i>srcPort</i>	Source port of the flow
<i>dstAddress</i>	Destination address of the flow.
<i>dstPort</i>	Destination port of the flow.
<i>Protocol</i>	IP Protocol. See <code>ippProto</code> values defined in <code>DetectorCommon.lua</code> . Example <code>DC.ippProto.tcp</code>

Returns:

DetectorFlowUserData - A userdata representing DetectorFlowUserData.

5 Appendix

5.1 Flow Flags

The following is a list of available flow-related flags that can be set, cleared, or viewed by detectors:

- `udpReversed (0x00400000)`
- `incompatible (0x00800000)`: Service protocol had incompatible client data
- `ignoreHost (0x01000000)`: Call service detection even if the host does not exist
- `ignoreTcpSeq (0x02000000)`: Ignore TCP state tracking
- `clientAppDetected (0x04000000)`: Finished with client app detection
- `gotBanner (0x08000000)`: Acquired a banner
- `notAService (0x10000000)`: Flow is a data connection, not a service
- `logUnknown (0x20000000)`: Log packets of the session
- `continue (0x40000000)`: Continue calling the routine after the service has been identified
- `serviceDetected (0x80000000)`: Service protocol was detected