PATHspider Documentation

Release 2.0

the pathspider authors

Jan 19, 2018

Contents

In today's Internet we see an increasing deployment of middleboxes. While middleboxes provide in-network functionality that is necessary to keep networks manageable and economically viable, any packet mangling — whether essential for the needed functionality or accidental as an unwanted side effect — makes it more and more difficult to deploy new protocols or extensions of existing protocols.

For the evolution of the protocol stack, it is important to know which network impairments exist and potentially need to be worked around. While classical network measurement tools are often focused on absolute performance values, PATHspider performs A/B testing between two different protocols or different protocol extensions to perform controlled experiments of protocol-dependent connectivity problems as well as differential treatment.

PATHspider is a framework for performing and analyzing these measurements, while the actual A/B tests can be easily customized. This documentation describes the architecture of PATHspider, the plugins available and how to use and develop and package the plugins.

CHAPTER 1

Table of Contents

1.1 Introduction

Network operators increasingly rely on in-network functionality to make their networks manageable and economically viable. These middleboxes make the end-to-end path for traffic more opaque by making assumptions about the traffic passing through them. This has led to an ossification of the Internet protocol stack: new protocols and extensions can be difficult to deploy when middleboxes do not understand them *[Honda11]*. PATHspider is a software measurement tool for active measurement of Internet path transparency to transport protocols and transport protocol extensions, that can generate raw data at scale to determine the size and shape of this problem.

The A/B testing measurement methodology used by PATHspider is simple: We perform connections from a set of observation points to a set of measurement targets using two configurations. A baseline configuration (A), usually a TCP connection using kernel default and no extensions, tests basic connectivity. These connections are compared to the experimental configuration (B), which uses a different transport protocol or set of TCP extensions. These connections are made as simultaneously as possible, to reduce the impact of transient network changes.

Since PATHspider 2.0, it is also possible to perform more than one B test optionally performing an A test between each B test in order to revalidate the path. This can also be used to "prime" the path for a follow up connection if it is desirable to have devices on the path hold state before performing the test.

PATHspider is a generalized version of the ecnspider tool, used in previous studies to probe the paths from multiple vantage points to web-servers *[Trammell15]* and to peer-to-peer clients *[Gubser15]* for failures negotiating Explicit Congestion Notification (ECN) *[RFC3168]* in TCP.

As a generalized tool for controlled experimental A/B testing of path impairment, PATHspider fills a gap in the existing Internet active measurement software ecosystem. Existing active measurement platforms, such as RIPE Atlas *[RIPEAtlas]*, OONI *[Filasto12]*, or Netalyzr *[Kreibich10]*, measure absolute performance and connectivity between a pair of endpoints under certain conditions. The results obtainable from each of these can be compared to each other to simulate A/B testing. However, the measurement data from these platforms provide a less controlled view than can be achieved with PATHspider, given coarser scheduling of measurements in each state.

Given PATHspider's modular design and implementation in Python, plugins to perform measurements for any transport protocol or extension are easy to build and can take advantage of the rich Python library ecosystem, including high-level application libraries, low-level socket interfaces, and packet forging tools such as Scapy.

1.1.1 Architecture

The PATHspider architecture has four components, illustrated in the diagram below the *configurator*, the *workers*, the *observer* and the *combiner*. Each component is implemented as one or more threads, launched when PATHspider starts.

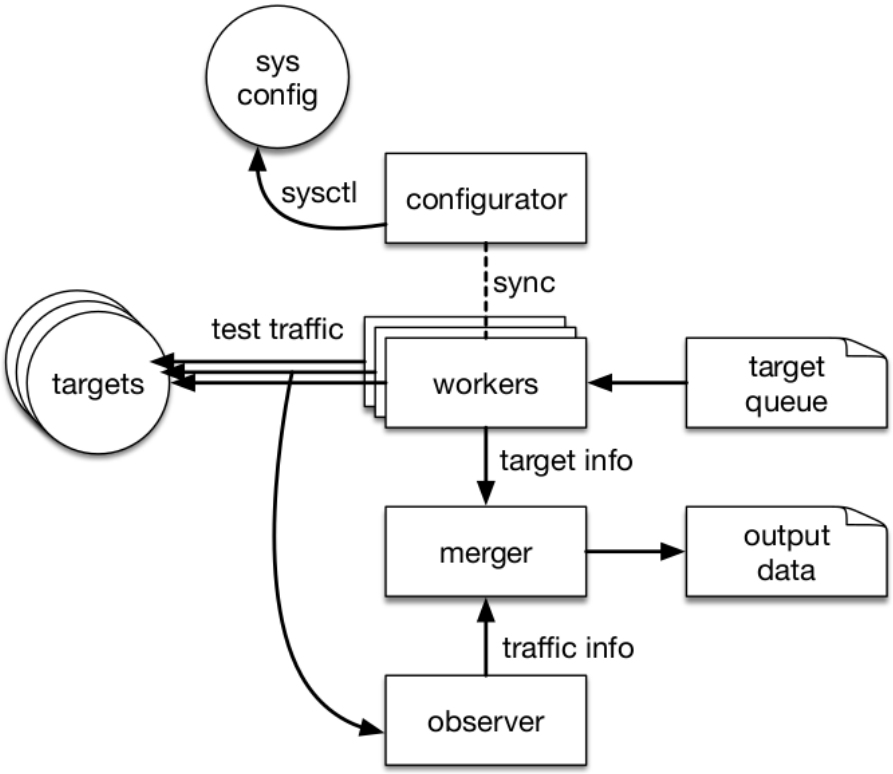

Fig. 1.1: An overview of the PATHspider architecture

For each target hostname and/or address, with port numbers where appropriate, PATHspider enqueues a job, to be distributed amongst the worker threads when available. Each worker performs one connection with the "A" configuration and one connection with the "B" configuration. The "A" configuration will always be connected first and serves as the base line measurement, followed by the "B" configuration. This allows detection of hosts that do not respond rather than failing as a result of using a particular transport protocol or extension. These sockets remain open for a post-connection operation.

Some transport options require a system-wide parameter change, for example enabling ECN in the Linux kernel. This requires locking and synchronisation. Using semaphores, the configurator waits for each worker to complete an operation and then changes the state to perform the next batch of operations. This process cycles continually until no more jobs remain.

In a typical experiment, multiple workers (on the order of hundreds) are active, since much of the time in a connection test is spent waiting for an answer from the target or a timeout to fire. Where it is possible to peform the tests without a system-wide configuration it is possible to disable the semaphores to increase the speed of the test.

In addition, packets are separately captured for analysis by the observer using Python bindings for libtrace. First, the observer assigns each incoming packet to a flow based on the source and destination addresses, as well as the TCP, UDP or SCTP ports when available. The packet and its associated flow are then passed to a function chain. The functions in this chain may be simple functions, such as counting the number of packets or octets seen for a flow, or more complex functions, such as recording the state of flags within packets and analysis based on previously

observed packets in the flow. For example, a function may record both an ECN negotiation attempt and whether the host successfully negotiated use of ECN.

Path conditions are generated for the path to each target to determine whether or not connectivity breakage has occured, or other conditions that may lead to more subtle breakage.

1.1.2 Extensibility

PATHspider plugins are built by extending an abstract class that implements the core behaviour, with functions for the configurator, workers, observer, and merger. There are three main abstract classes that can be extended by plugins: *pathspider.sync.SynchronizedSpider*, *pathspider.desync.DesynchronizedSpider* and *pathspider.forge.ForgeSpider*.

Depending on the type of plugin being created, these abstract classes are extended to include logic for generating the active measurement traffic.

Plugins can implement arbitrary functions for the observer function chain, or reuse library functions for some functionality. These track the state of flows and build flow records for different packet classes: The first chain handles setup on the first packet of a new flow. Separate chains for IP, TCP and UDP packets allow different behaviours based on the IP version and transport protocol.

The final plugin function is the combiner function. This takes a list of merged job records and flow records to produce "path conditions" before passing the final job record back to PATHspider for output.

1.2 Installation

1.2.1 Debian GNU/Linux

Note: If there has not been much time since the release, the Debian packages for the latest version may not yet be available.

PATHspider is packaged for Debian and packages are made available for the testing and stable-backports distributions. If you are running Debian stable, ensure that you have enabled the stable-backports repository in your apt sources.

To install PATHspider, simply run:

```
sudo apt install pathspider
```

1.2.2 Vagrant

Warning: Depending on the set up of your Vagrant virtualization provider, some tests may be affected. It is wise to test against known configurations to ensure that your networking set up has a clear path to the Internet before running larger measurement campaigns.

On systems other than Linux systems, you may use Vagrant to run PATHspider. This may also be useful during development. A Vagrantfile is provided that will create a Debian-based virtual machine with all the PATHspider dependencies installed.

In the virtual machine, the PATHspider code will be copied to `/home/vagrant/pathspider`. To improve compatibility across platforms, this is not synchronized with the repository outside of the Vagrant image. Expert users may

edit the `Vagrantfile` to achieve this. PATHspider is installed in development mode, meaning that this is also the location of the PATHspider code that will be run when running the `/usr/bin/pspdr` command inside the virtual machine.

Assuming that you have Vagrant and a virtualisation provider (e.g. VirtualBox) installed, you can get started with:

```
vagrant up
vagrant ssh
```

Depending on the speed of your Internet connection, this may take a long time.

1.2.3 Source

Warning: PATHspider 2.0 depends on pycurl >= 7.43.0.1, released on the 7th December 2017. If you have errors when running PATHspider similar to `AttributeError: module 'pycurl' has no attribute 'CONNECT_TO'` then it is most likely the case that your version of pycurl is too old.

If you are working from the source distribution (e.g. cloned git repository) then you will need to install the required dependencies. On Debian GNU/Linux, assuming you have the stable-backports repository enabled if you are running stable:

```
sudo apt build-dep pathspider
```

Note: This will install both the runtime and the build dependencies required for PATHspider, its testsuite and its documentation.

On other platforms, you may install most of the dependencies required via pip:

```
pip install -r requirements.txt
```

Unfortunately, python-libtrace is not available on PyPI and so must be installed seperately. You will also need to ensure that for both pycurl and python-libtrace you have the build dependencies available as these are compiled CPython modules.

If you wish to build the documentation from source or to use the testsuite, and you are installing your dependencies via pip, you will also need the following dependencies:

```
pip install -r requirements_dev.txt
```

With the dependencies installed, you can install PATHspider with:

```
python3 setup.py install
```

1.3 Command Line Usage Overview

You can run PATHspider from the command line. In order for the Observer to work, you will need permissions to capture raw packets from the network interface. You may also need elevated privileges when generating traffic using raw sockets or to modify the local TCP/IP stack. This will require you to use `sudo` or equivalent in order to run PATHspider if you are not logged in as the root user.

```
# pspdr --help
usage: pspdr [-h] [--verbose] COMMAND ...

PATHspider will spider the paths.

optional arguments:
  -h, --help  show this help message and exit
  --verbose   Enable verbose logging

Commands:
    filter    Pre-process a target list
    measure   Perform a PATHspider measurement
    observe   Passively observe network traffic
    test      Run the built in test suite

Spider safely!
```

1.3.1 Performing Active Measurement

PATHspider provides the "measure" command to perform active traffic generation and observation of that traffic for path transparency measurement. Based on the observations made, paths are assigned conditions such as *ecn.connectivity.works* indicating that the use of ECN does not cause connectivity impairment between the vantage point and the particular target.

It is possible to enable the output of flow records along with the derived observations using the `--output-flows` flag. This will generate considerably more output and so is disabled by default.

You may specify input and output files using flags, however by default these are set to be stdin and stdout and so you can, and are recommended to, use shell redirection instead. To see output as it is written to the file, you can pipe the output to `tee` to print it on the screen while also saving it to a file.

You will be required to set your interface name and PATHspider will not start if it detects that the chosen interface is not active.

```
# pspdr measure --help
usage: pspdr measure [-h] [-i INTERFACE] [-w WORKERS] [--input INPUTFILE]
                     [--output OUTPUTFILE] [--output-flows]
                     PLUGIN ...

optional arguments:
  -h, --help            show this help message and exit
  -i INTERFACE, --interface INTERFACE
                        The interface to use for the observer. (Default: eth0)
  -w WORKERS, --workers WORKERS
                        Number of workers to use. (Default: 100)
  --input INPUTFILE     A file containing a list of PATHspider jobs. Defaults
                        to standard input.
  --output OUTPUTFILE   The file to output results data to. Defaults to
                        standard output.
  --output-flows        Include flow results in output.

Plugins:
  The following plugins are available for use:

    tfo                 TCP Fast Open
    ecn                 Explicit Congestion Notification
    h2                  HTTP/2
```

```
    dscp                Differentiated Services Codepoints
    dnsresolv           Simple Input List DNS Resolver
    udpopts             UDP Options Trailer
    udpzero             UDP Zero Checksum

Spider safely!
```

Quickstart Example

You can run a small study using the ECN plugin and the included `webtest.ndjson` file to measure path transparency to ECN for a small selection of web servers and save the results in `results.ndjson` (ensure to change the interface name to match an active interface on your machine):

```
pspdr measure -i eth0 ecn </usr/share/doc/pathspider/examples/webtest.ndjson >results.
↪ndjson
```

Note: If you've not installed PATHspider from apt, you will find the webinput.ndjson example input file in the examples folder of the source distribution.

1.3.2 Performing Passive Observation

PATHspider provides the "observe" command to perform passive traffic observation for path transparency measurement. In this version of PATHspider we do not attempt to determine path conditions during passive observation, and instead only output flow records. This may change in future versions of PATHspider.

You can list the available chains with `--list-chains` and then select any number of chains that you would like to use. It is recommended that you include the *basic* chain as this will add the IP addresses and port numbers to the flow records.

You may specify the output file using a flag, however by default this is set to be `/dev/stdout` and so you can, and are recommended to, use shell redirection instead. To see output as it is written to the file, you can pipe the output to `tee` to print it on the screen while also saving it to a file. You will be required to set your interface name and PATHspider will not start if it detects that the chosen interface is not active.

It is also possible to perform offline analysis of a PCAP file using the "observe" command. Instead of an interface name, pass the name of the pcap file to `-i` instead. The PCAP file must have a `.pcap` extension to be recognised.

```
usage: pspdr observe [-h] [--list-chains] [-i INTERFACE] [--output OUTPUTFILE]
                     [chains [chains ...]]

positional arguments:
  chains                Observer chains to use

optional arguments:
  -h, --help            show this help message and exit
  --list-chains         Prints a list of available chains
  -i INTERFACE, --interface INTERFACE
                        The interface to use for the observer. (Default: eth0)
  --output OUTPUTFILE   The file to output results data to. Defaults to
                        standard output.
```

Quickstart Example

You can observe network traffic passively to perform observations without actively generating traffic. In this case no input file is needed.

```
pspdr observe -i eth0 basic tcp ecn >results.ndjson
```

1.3.3 Data Formats

PATHspider uses newline delimited JSON (ndjson) for both the input and output format. The ndjson format gives flexibility in the contents of the data as different tests may require data to remain associated with jobs so that it can be present in the final output (the Alexa ranking of a webserver, for example), or used as part of the test (running tests against authoritative DNS servers and needing to know a domain for which the server should be authoritative).

Input Format

At a minimum, each job should contain an IP address in a `dip` field. Depending on the plugin in use, more details may be required. Refer to the documentation for the specific plugin for more information.

Output Format

For each job, the output JSON dictionary will contain the original job information, a computed path using available information and a set of conditions seen for the path as generated by the plugins.

With `--output-flows` enabled, PATHspider's output will include an additional field in the JSON dictionary for each job containing an array of flow records, one for each configuration. Usually one record will be for the baseline (A) connection, and one for the experimental (B) connection. These JSON records contain the original job information, any information added by the connection functions and any information added by the Observer.

Additionally, internal information may be retained:

Key	Description
config	0 for baseline, 1..n for experimental
spdr_state	0 = OK, 1 = TIMEOUT, 2 = FAILED, 3 = SKIPPED

For detail on the values in individual plugins, see the section for that plugin later in this documentation.

1.4 Active Measurement Plugins

A number of plugins ship with the PATHspider distribution. You can find documentation for them here:

1.4.1 DSCP Plugin

Differentiated services or DiffServ *[RFC2474]* is a networking architecture that specifies a simple, scalable and coarse-grained mechanism for classifying and managing network traffic and providing quality of service (QoS) on modern IP networks. DiffServ can, for example, be used to provide low-latency to critical network traffic such as voice or streaming media while providing simple best-effort service to non-critical services such as web traffic or file transfers.

DiffServ uses a 6-bit differentiated services code point (DSCP) in the 8-bit differentiated services field (DS field) in the IP header for packet classification purposes. The DS field and ECN field replace the outdated IPv4 TOS field. *[RFC3260]*

The DSCP plugin for PATHspider aims to detect breakage in the Internet due to the use of a non-zero DSCP codepoint.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the DSCP plugin, specify `dscp` as the plugin to use on the command-line:

```
pspdr measure -i eth0 dscp </usr/share/doc/pathspider/examples/webtest.ndjson >
↪results.ndjson
```

This will run two HTTP GET request connections over TCP for each job input, one with the DSCP set to zero (best-effort) and one with the DSCP set to 46 (expedited forwarding). If you would like to specify the code point for use on the experimental flow, you may do this with the `--codepoint` option. For example, to use 42:

```
pspdr measure -i eth0 dscp --codepoint 42 </usr/share/doc/pathspider/examples/webtest.
↪ndjson >results.ndjson
```

Supported Connection Modes

This plugin supports the following connection modes:

- http - Performs a GET request
- tcp - Performs only a TCP 3WHS
- dnsudp - Performs a DNS query using UDP
- dnstcp - Performs a DNS query using TCP

To use an alternative connection mode, add the `--connect` argument to the invocation of PATHspider:

```
pspdr measure -i eth0 dscp --connect tcp </usr/share/doc/pathspider/examples/webtest.
↪ndjson >results.ndjson
```

Output Conditions

The following conditions are generated for the DSCP plugin:

dscp.X.connectivity.Y

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity was successful using codepoint X validated against a connection using codepoint 0 (zero).

Y may have the following values:

- works - Both connections succeeded

- broken - Baseline connection succeeded where experimental connection failed
- offline - Both connections failed
- transient - Baseline connection failed where experimental connection succeeded (this can be used to give an indication of transient failure rates included in the "broken" set)

dscp.X.replymark:

For each connection that was observed to have a response by PATHspider, a condition is generated to show values of codepoints set on response packets when codepoint X was set.

Notes

- DSCP marking is performed using the `mangle` table in `iptables`. The `config_zero` function will flush this table. PATHspider makes no guarantees the the configuration state is consistent once it has been set, though you can use the forward path markings in the output to validate the results within a reasonably high level of certainty that everything behaved correctly.

1.4.2 ECN Plugin

Explicit Congestion Notification (ECN) is an extension to the Internet Protocol and to the Transmission Control Protocol. *[RFC3168]* ECN allows end-to-end notification of network congestion without dropping packets. ECN is an optional feature that may be used between two ECN-enabled endpoints when the underlying network infrastructure also supports it.

Conventionally, TCP/IP networks signal congestion by dropping packets. When ECN is successfully negotiated, an ECN-aware router may set a mark in the IP header instead of dropping a packet in order to signal impending congestion. The receiver of the packet echoes the congestion indication to the sender, which reduces its transmission rate as if it detected a dropped packet.

Rather than responding properly or ignoring the bits, some outdated or faulty network equipment has historically dropped or mangled packets that have ECN bits set. As of 2015, measurements suggested that the fraction of web servers on the public Internet for which setting ECN prevents network connections had been reduced to less than 1%. *[Trammell15]*

The ECN plugin for PATHspider aims to detect breakage in the Internet due to the use of ECN.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the ECN plugin, specify `ecn` as the plugin to use on the command-line:

```
pspdr measure -i eth0 ecn </usr/share/doc/pathspider/examples/webtest.ndjson >results.
↪ndjson
```

This will run two TCP connections for each job input, one with ECN disabled in the kernel TCP/IP stack and one with ECN enabled in the kernel TCP/IP stack.

Supported Connection Modes

This plugin supports the following connection modes:

- http - Performs a GET request
- https - Performs a GET request using HTTPS
- tcp - Performs only a TCP 3WHS
- dnstcp - Performs a DNS query using TCP

To use an alternative connection mode, add the `--connect` argument to the invocation of PATHspider:

```
pspdr measure -i eth0 ecn --connect tcp </usr/share/doc/pathspider/examples/webtest.
↪ndjson >results.ndjson
```

Output Conditions

The following conditions are generated for the ECN plugin:

ecn.connectivity.Y

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity was successful using ECN against a connection not using ECN.

Y may have the following values:

- works - Both connections succeeded
- broken - Baseline connection succeeded where experimental connection failed
- offline - Both connections failed
- transient - Baseline connection failed where experimental connection succeeded (this can be used to give an indication of transient failure rates included in the "broken" set)

ecn.negotiation.Y

For each experimental connection that was observed to have a response by PATHspider, a condition is generated to show whether or not ECN negotiation succeded between the two hosts.

Y may have the following values:

- succeeded - ECN negotiation succeeded
- reflected - ECN negotiation failed, with both ECE and CWR set on reply SYN
- failed - ECN negotiation failed

ecn.ipmark.X.Y

For each connection that was observed by PATHspider, a condition is generated to record the ECN marks seen. Y has two possible values, "seen" or "not_seen", corresponding to whether or not mark X was encountered.

X may have the following values:

- ect0 - ECN Capable Transport (0)

- ect1 - ECN Capable Transport (1)
- ce - Congestion Experienced

Notes

- ECN behaviour is implemented by the host kernel for PATHspider, and is switched by a `sysctl` call. PATHspider makes no guarantees the the configuration state is consistent once it has been set, though you can use the forward SYN flags in the output to validate the results within a reasonably high level of certainty that everything behaved correctly.

1.4.3 Evil Bit Plugin

The Evil Bit refers to the unused high-order bit of the IP fragment offset field in the IP header. It was defined in RFC3514 on the 1st of April 2003.

The Evil Bit plugin for PATHspider aims to detect breakage in the Internet due to the use of reserved bit in the IP fragment offset field.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the EvilBit plugin, specify `evilbit` as the plugin to use on the command-line:

```
pspdr measure -i eth0 evilbit </usr/share/doc/pathspider/examples/webtest.ndjson >
↪results.ndjson
```

This will run two TCP connections for each job input, one with the evil bit not set and one with the evil bit set (indicating the packet's malicious intent).

Supported Connection Modes

This plugin supports the following connection modes:

- tcp - Performs only a TCP 3WHS
- dnsudp - Performs a DNS query using UDP

To use an alternative connection mode, add the `--connect` argument to the invocation of PATHspider:

```
pspdr measure -i eth0 evilbit --connect tcp </usr/share/doc/pathspider/examples/
↪webtest.ndjson >results.ndjson
```

Output Conditions

The following conditions are generated for the evilbit plugin:

evilbit.X.connectivity.Y

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity with the evil bit set was successful validated against a connection without the evil bit set.

Y may have the following values:

- works - Both connections succeeded
- broken - Baseline connection succeeded where experimental connection failed
- offline - Both connections failed
- transient - Baseline connection failed where experimental connection succeeded (this can be used to give an indication of transient failure rates included in the "broken" set)

evilbit.mark.X

A condition is generated to show whether the evil bit was set on the return path for the experimental connection. X can have two values, "seen" or "not_seen".

Notes

- The evil bit is set using packet forging library Scapy. Due to the lack of a TCP state machine, connection types such as HTTP are not available.

1.4.4 H2 Plugin

HTTP/2 (originally named HTTP/2.0) is a major revision of the HTTP network protocol used in the Internet.

The HTTP Upgrade mechanism is used to establish HTTP/2 starting from plain HTTP. The client starts a HTTP/1.1 connection and sends "Upgrade: h2c" header. If the server supports HTTP/2, it replies with HTTP 101 Switching Protocol status code. The HTTP Upgrade mechanism is used only for cleartext HTTP2 (h2c). In the case of HTTP2 over TLS (h2), the ALPN TLS protocol extension is used instead.

The h2 plugin for PATHspider aims to detect breakage in the Internet due to the use of HTTP/2.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the H2 plugin, specify `h2` as the plugin to use on the command-line:

```
pspdr measure -i eth0 h2 </usr/share/doc/pathspider/examples/webtest.ndjson >results.
↪ndjson
```

This will run two HTTP GET request connections over TCP for each job input, one without requesting an upgrade and one requesting an upgrade to HTTP/2.

Supported Connection Modes

This plugin supports the following connection modes:

- http - Performs a GET request
- https - Performs a GET request using HTTPS

To use an alternative connection mode, add the `--connect` argument to the invocation of PATHspider:

```
pspdr measure -i eth0 h2 --connect https </usr/share/doc/pathspider/examples/webtest.
↪ndjson >results.ndjson
```

Output Conditions

The following conditions are generated for the H2 plugin:

h2.connectivity.X

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity was successful using HTTP/2 against a connection using HTTP/1.1.

Y may have the following values:

- works - Both connections succeeded
- broken - Baseline connection succeeded where experimental connection failed
- offline - Both connections failed
- transient - Baseline connection failed where experimental connection succeeded (this can be used to give an indication of transient failure rates included in the "broken" set)

h2.upgrade.X

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not an upgrade request to HTTP/2 was successful. X can have two values, "success" or "failed".

Notes

- The H2 plugin uses cURL options to set the HTTP version to be used for the request and uses the version negotiation techniques built-in to cURL.

1.4.5 TCP Maximum Segment Size Plugin

The Transmission Control Protocol (TCP) Maximum Segment Size (MSS) option was one of the TCP options defined in the very first specification for TCP *[RFC793]*:

> If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This field must only be sent in the initial connection request (i.e., in segments with the SYN control bit set). If this option is not used, any segment size is allowed.

Due to the prevalent blocking of ICMP throughout the Internet (if you do this, please stop!), path maximum transmission unit (PMTU) discovery often fails to correctly determine the MTU that can safely be used between two hosts. As an alternative strategy, routers can rewrite the TCP MSS option present on SYN packets to ensure that the MSS seen by the receiving end of the packets is not greater than that which is supported on the links connected to that router.

The MSS plugin for PATHspider aims to discover the value of MSS that is received when connecting to hosts using TCP and compares this to the local MTU to determine if the received MSS is lower (possibly indicating the clamping behaviour described above), equal or greater (possibly indicating an unsafe MSS) than the local MSS.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the MSS plugin, specify `mss` as the plugin to use on the command-line:

```
pspdr measure -i eth0 mss </usr/share/doc/pathspider/examples/webtest.ndjson >results.
↪ndjson
```

This will open a TCP connection for each job input, recording the received MSS for each reply.

Supported Connection Modes

This plugin supports the following connection modes:

- tcp - Performs a TCP connection
- http - Performs a GET request
- https - Performs a GET request using HTTPS
- dnstcp - Performs a DNS query using TCP

To use an alternative connection mode, add the `--connect` argument to the invocation of PATHspider:

```
pspdr measure -i eth0 mss --connect tcp </usr/share/doc/pathspider/examples/webtest.
↪ndjson >results.ndjson
```

Output Conditions

The following conditions are generated for the MSS plugin:

mss.connectivity.Y

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity was successful using TFO validated against a connection not using TFO.

Y may have the following values:

- online - The connection succeeded
- offline - The connection failed

mss.option.X.value:Y

For each connection that was observed to have a response by PATHspider and observed to have an MSS option in the TCP header, a condition is generated to show the value of the MSS option.

X can have two values, "local" or "remote", idicating whther the option was sent locally or received from the remote target (possibly having been rewritten on the path). Y is the value of the option.

mss.option.received.X

For each connection that was observed to have a response by PATHspider, a condition is generated to show whether the MSS option was absent or present. If present, it will be compared to the local MSS. X can have the following values:

- absent - The response from the remote target did not contain an MSS option in the TCP header.
- unchanged - The MSS option received from the remote target contained the same value as the local MSS.
- inflated - The MSS option received from the remote target contained a greater MSS than the local MSS.
- deflated - The MSS option received from the remote target contained a lower MSS than the local MSS.

1.4.6 UDP Zero Checksum Plugin

UDP uses a 16-bit field to store a checksum for data integrity. The UDP checksum field *[RFC768]* is calculated using information from the pseudo-IP header, the UDP header, and the data is padded at the end if necessary to make a multiple of two octets. The checksum is optional when using IPv4, and if unused a UDP checksum field carrying all zeros indicates the transmitter did not compute the checksum.

The UDPZero plugin for PATHspider aims to detect breakage in the Internet due to the use a zero-checksum field.

Usage Example

Note: The path given to the example list of web servers is taken from a Debian GNU/Linux installation and may differ on your computer. These are the same examples that can be found in the *examples/* directory of the source distribution.

To use the UDPZero plugin, specify `udpzero` as the plugin to use on the command-line:

```
pspdr measure -i eth0 udpzero </usr/share/doc/pathspider/examples/webtest.ndjson >
↪results.ndjson
```

This will run two DNS request connections over UDP for each job input, one with the checksum field unmodified and one with the checksum field set to all zeros.

Supported Connection Modes

This plugin supports the following connection modes:

- dnsudp - Performs a DNS query using UDP

Output Conditions

The following conditions are generated for the UDPZero plugin:

udpzero.connectivity.Y

For each connection that was observed by PATHspider, a connectivity condition will be generated to indicate whether or not connectivity was successful using UDP zero-checksum validated against a connection with the calculated checksum left intact.

Y may have the following values:

- works - Both connections succeeded
- broken - Baseline connection succeeded where experimental connection failed
- offline - Both connections failed
- transient - Baseline connection failed where experimental connection succeeded (this can be used to give an indication of transient failure rates included in the "broken" set)

Notes

- Setting the UDP checksum field to all zeros is performed using Python library Scapy.

1.4.7 3rd-Party Plugins

You will be able to list the 3rd-party plugins installed by running:

```
pathspider --help
```

There is no need to register 3rd-party plugins with PATHspider before use, they will be automatically detected once they are installed.

1.5 Passive Observation

The passive observation modules (sometimes referred to as "observer chains") are used by the active measurement plugins, but can also be used independently. At this time, it is not possible to create new passive observation modules as plugins unless they are part of an active measurement plugin.

A number of modules ship with the PATHspider distribution. You can find their documentation here:

1.5.1 Basic Chain

This module contains the BasicChain flow analysis chain which can be used by PATHspider's Observer for recording source and destination addresses and packet/octet counts.

class pathspider.chains.basic.**BasicChain**
This flow analysis chain records details from the TCP/IP headers.

Field Name	Type	Meaning
dip	str	Layer 3 (IPv4/IPv6) source address
sp	int	Layer 4 (TCP/UDP) source port
dp	int	Layer 4 (TCP/UDP) destination port
pkt_fwd	int	A count of the number of packets seen in the forward direction
pkt_rev	int	A count of the number of packets seen in the reverse direction
oct_fwd	int	A count of the number of octets seen in the forward direction
oct_rev	int	A count of the number of octets seen in the reverse direction

new_flow(*rec*, *ip*)
New flow function that sets up basic flow information

1.5.2 DNS Chain

This module contains the DNSChain flow analysis chain which can be used by PATHspider's Observer for recording Domain Name System *[RFC1035]* details.

class pathspider.chains.dns.**DNSChain**
This flow analysis chain records details from Domain Name System application data.

Field Name	Type	Meaning
dns_response_valid	bool	The flow contained a valid DNS response

new_flow(*rec*, *ip*)
For a new flow, all fields will be initialised to `False`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

tcp(*rec*, *tcp*, *rev*)
Records DNS details from TCP segment.

DNS Response If the packet contains a payload, an attempt is made to parse it and if successful the `dns_response_valid` field is set to `True` if it was a response (not a query).

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the TCP packet that was observed to be part of this flow
- **rev** (*bool*) – `True` if the packet was in the reverse direction, `False` if in the forward direction

Returns `False` if a valid DNS response has been seen, otherwise `True`

Return type bool

udp(*rec*, *udp*, *rev*)
Records DNS details from UDP datagram.

DNS Response If the packet contains a payload, an attempt is made to parse it and if successful the `dns_response_valid` field is set to `True` if it was a response (not a query).

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the UDP packet that was observed to be part of this flow
- **rev** (*bool*) – `True` if the packet was in the reverse direction, `False` if in the forward direction

Returns `False` if a valid DNS response has been seen, otherwise `True`

Return type bool

1.5.3 DSCP Chain

This module contains the DSCPChain flow analysis chain which can be used by PATHspider's Observer for recording Differentiated Services *[RFC2474]* details.

class `pathspider.chains.dscp.`**DSCPChain**

This flow analysis chain records details of the Differentiated Services Field in the IP header.

Field Name	Type	Meaning
`dscp_mark_syn_fwd`	int	The value of the Differentiated Services codepoint seen on a TCP SYN packet in the forward direction
`dscp_mark_data_fwd`	int	The value of the Differentiated Services codepoint seen on a non-TCP packet or a TCP packet with a payload in the forward direction
`dscp_mark_syn_rev`	int	The value of the Differentiated Services codepoint seen on a TCP SYN packet in the reverse direction
`dscp_mark_data_rev`	int	The value of the Differentiated Services codepoint seen on a non-TCP packet or a TCP packet with a payload in the reverse direction

ip4 (*rec*, *ip*, *rev*)

Records DSCP markings from an IPv4 header.

DSCP Marking For the first TCP SYN packet and the first non-TCP packet or TCP packet with a payload, the DSCP value will be recorded in the relevant field.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always `True`

Return type bool

ip6 (*rec*, *ip*, *rev*)

Records DSCP markings from an IPv6 header.

DSCP Marking For the first TCP SYN packet and the first non-TCP packet or TCP packet with a payload, the DSCP value will be recorded in the relevant field.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip6*) – the IPv6 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always `True`

Return type bool

new_flow(*rec*, *ip*)

For a new flow, all fields will be initialised to `None`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

1.5.4 ECN Chain

This module contains the ECNChain flow analysis chain which can be used by PATHspider's Observer for recording Explicit Congestion Notification *[RFC3168]* details.

class pathspider.chains.ecn.**ECNChain**

This flow analysis chain records details of ECN markings in the IP header.

Field Name	Type	Meaning
ecn_ect0_syn_fwd	bool	An ECT0 mark was seen in the forward direction on a TCP SYN packet
ecn_ect1_syn_fwd	bool	An ECT1 mark was seen in the forward direction on a TCP SYN packet
ecn_ce_syn_fwd	bool	An CE mark was seen in the forward direction on a TCP SYN packet
ecn_ect0_data_fwd	bool	An ECT0 mark was seen in the forward direction on a TCP packet with a payload or a non-TCP packet
ecn_ect1_data_fwd	bool	An ECT1 mark was seen in the forward direction on a TCP packet with a payload or a non-TCP packet
ecn_ce_data_fwd	bool	An CE mark was seen in the forward direction on a TCP packet with a payload or a non-TCP packet
ecn_ect0_syn_rev	bool	An ECT0 mark was seen in the reverse direction on a TCP SYN packet
ecn_ect1_syn_rev	bool	An ECT1 mark was seen in the reverse direction on a TCP SYN packet
ecn_ce_syn_rev	bool	An CE mark was seen in the reverse direction on a TCP SYN packet
ecn_ect0_data_rev	bool	An ECT0 mark was seen in the reverse direction on a TCP packet with a payload or a non-TCP packet
ecn_ect1_data_rev	bool	An ECT1 mark was seen in the reverse direction on a TCP packet with a payload or a non-TCP packet
ecn_ce_data_rev	bool	An CE mark was seen in the reverse direction on a TCP packet with a payload or a non-TCP packet

ip4(*rec*, *ip*, *rev*)

Records ECN markings from an IPv4 header.

ECN Marking If an ECT0, ECT1 or CE mark is seen in the IPv4 header, the relevant field will be set to `True`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always `True`

Return type bool

ip6(*rec*, *ip*, *rev*)
Records ECN markings from an IPv6 header.

ECN Marking If an ECT0, ECT1 or CE mark is seen in the IPv6 header, the relevant field will be set to `True`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip6*) – the IPv6 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always `True`

Return type bool

new_flow(*rec*, *ip*)
For a new flow, all fields will be initialised to `False`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

1.5.5 Evil Bit Chain

This module contains the EvilChain analysis chain which can be used by PATHspider's Observer for recording Evil Bit connectivity [RFC3514] details.

class pathspider.chains.evil.**EvilChain**

Field Name	Type	Meaning
`evilbit_syn_fwd`	bool	True if the evil bit was set in the IP header for a TCP SYN packet in the forward direction, false otherwise
`evilbit_syn_fwd`	bool	True if the evil bit was set in the IP header for a TCP SYN packet in the reverse direction, false otherwise
`evilbit_data_fwd`	bool	True if the evil bit was set in the IP header for a non-TCP packet in the forward direction, false otherwise
`evilbit_data_rev`	bool	True if the evil bit was set in the IP header for a non-TCP packet in the reverse direction, false otherwise

ip4 (*rec*, *ip*, *rev*)
Records evil bit markings from an IPv4 header.

Evil Bit Marking For either TCP_SYN packets or non-TCP or TCP with payload packets the relevant field will record whether the Evil Bit was set.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always `True`

Return type bool

new_flow (*rec*, *ip*)
For a new flow, all fields will be initialised to `None`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

1.5.6 ICMP Chain

This module contains the ICMPChain flow analysis chain which can be used by PATHspider's Observer for recording ICMPv4 *[RFC792]* and ICMPv6 *[RFC4443]* details.

pathspider.chains.icmp.**ICMP4_TTLEXCEEDED = 11**
ICMPv4 Message Type - TTL Exceeded

pathspider.chains.icmp.**ICMP4_UNREACHABLE = 3**
ICMPv4 Message Type - Unreachable

pathspider.chains.icmp.**ICMP6_TTLEXCEEDED = 3**
ICMPv6 Message Type - Time Exceeded

pathspider.chains.icmp.**ICMP6_UNREACHABLE = 1**
ICMPv6 Message Type - Unreachable

class pathspider.chains.icmp.**ICMPChain**

This flow analysis chain records details of ICMP messages in the flow record. It will record when a message of certain types have been seen during a flow.

Field Name	Type	Meaning
icmp_unreachable	bool	An ICMP unreachable message was seen in the reverse direction

icmp4(*rec*, *ip*, *q*, *rev*)

Records ICMPv4 details.

ICMPv4 Unreachable Messages Sets `icmp_unreachable` to `True` if an ICMP Unreachable message is seen in the reverse direction.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow and contained an ICMPv4 header
- **q** (*plt.ip*) – the ICMP quotation of the packet that triggered this message (if any)
- **rev** (*bool*) – `True` if the packet was in the reverse direction, `False` if in the forward direction

Returns `False` if an ICMP unreachable message has been observed, otherwise `True`

Return type bool

icmp6(*rec*, *ip6*, *q*, *rev*)

Records ICMPv6 details.

ICMPv6 Unreachable Messages Sets `icmp_unreachable` to `True` if an ICMP Unreachable message is seen in the reverse direction.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip6*) – the IPv6 packet that was observed to be part of this flow and contained an ICMPv6 header
- **q** (*plt.ip*) – the ICMP quotation of the packet that triggered this message (if any)
- **rev** (*bool*) – `True` if the packet was in the reverse direction, `False` if in the forward direction

Returns `False` if an ICMP unreachable message has been observed, otherwise `True`

Return type bool

new_flow(*rec*, *ip*)

For a new flow, all fields will be initialised to `False`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

1.5.7 TCP Maximum Segment Size Chain

This module contains the MSSChain flow analysis chain which can be used by PATHspider's Observer for recording TCP Maximum Segment Size details.

class pathspider.chains.mss.**MSSChain**

This flow analysis chain records details of the TCP Maximum Segment Size (MSS) option in the flow record. It will determine the length and value of the field if present in SYN packets.

Field Name	Type	Meaning
mss_len_fwd	int	Length of the MSS option field including kind and length in the forward direction.
mss_len_rev	int	Length of the MSS option field including kind and length in the reverse direction.
mss_value_fwd	int	Value of the MSS option field in the forward direction.
mss_value_rev	int	Value of the MSS option field in the reverse direction.

new_flow(*rec*, *ip*)

For a new flow, all fields will be initialised to None.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always True

Return type bool

tcp(*rec*, *tcp*, *rev*)

Records TCP Maximum Segment Size Details.

TCP Maximum Segment Size The TCP options will be parsed for the MSS option for all SYN packets. If the option is found, the length and value for the option will be recorded in the flow.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the TCP segment that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always True

Return type bool

1.5.8 TCP Chain

This module contains the TCPChain flow analysis chain which can be used by PATHspider's Observer for recording basic TCP *[RFC793]* behaviour details. This module also contains a helper function that may be used by chains for the parsing of TCP options and a number of useful TCP related constants that can be used to interpret the results added to flow records by TCPChain.

class pathspider.chains.tcp.**TCPChain**

This flow analysis chain records details of basic TCP behaviour in the flow record. It will determine when a 3WHS has completed and has simplified logic for determining when a TCP flow has completed.

Field Name	Type	Description
tcp_synflags_fwd	int	SYN flags seen in the forward direction
tcp_synflags_rev	int	SYN flags seen in the reverse direction
tcp_fin_fwd	bool	At least one FIN flag was seen in the forward direction
tcp_fin_rev	bool	At least one FIN flag was seen in the reverse direction
tcp_rst_fwd	bool	At least one RST flag was seen in the forward direction
tcp_rst_rev	bool	At least one RST flag was seen in the reverse direction
tcp_connected	bool	The 3WHS completed

new_flow(*rec*, *ip*)

For a new flow, all fields will be initialised to `False` except `tcp_synflags_*` which will be set to `None`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

tcp(*rec*, *tcp*, *rev*)

Records basic TCP behaviour details.

SYN Flags This will record the SYN flags observed in each direction. These will not be recorded again if there are futher segments in the flow with a SYN bit set, the first SYN observed wins.

FIN and RST Flags If a segment has the FIN or RST flags, the relevant fields are set to true.

3WHS If a SYN was observed in the forward direction, and a SYNACK in the reverse direction and the segment passed is an ACK in the forward direction then `tcp_connected` will be set to True.

Flow Completion If a FIN has been observed in one direction and this segment contains a FIN in the other direction, a flow is considered complete. If a RST has been observed in either direction, a flow is considered complete.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the TCP segment that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

pathspider.chains.tcp.**TCP_ACK = 16**

TCP Flag - ACK

pathspider.chains.tcp.**TCP_CWR = 128**
TCP Flag - CWR

pathspider.chains.tcp.**TCP_ECE = 64**
TCP Flag - ECE

pathspider.chains.tcp.**TCP_FIN = 1**
TCP Flag - FIN

pathspider.chains.tcp.**TCP_PSH = 8**
TCP Flag - PSH

pathspider.chains.tcp.**TCP_RST = 4**
TCP Flag - RST

pathspider.chains.tcp.**TCP_SA = 18**
TCP Flags - SYN and ACK

pathspider.chains.tcp.**TCP_SAE = 82**
TCP Flags - SYN, ACK, ECE

pathspider.chains.tcp.**TCP_SAEC = 210**
TCP Flags - SYN, ACK, ECE and CWR

pathspider.chains.tcp.**TCP_SEC = 194**
TCP Flags - SYN, ACK and ECE

pathspider.chains.tcp.**TCP_SYN = 2**
TCP Flag - SYN

pathspider.chains.tcp.**TCP_URG = 32**
TCP Flag - URG

pathspider.chains.tcp.**TO_EOL = 0**
TCP Option - End of options list

pathspider.chains.tcp.**TO_EXID_FASTOPEN = (249, 137)**
TCP Option Experiment ID - TCP Fast Open

pathspider.chains.tcp.**TO_EXPA = 254**
TCP Option - Experimental Option A

pathspider.chains.tcp.**TO_EXPB = 255**
TCP Option - Experimental Option B

pathspider.chains.tcp.**TO_FASTOPEN = 34**
TCP Option - TCP Fast Open Cookie

pathspider.chains.tcp.**TO_MPTCP = 30**
TCP Option - Multipath TCP

pathspider.chains.tcp.**TO_MSS = 2**
TCP Option - Maximum Segment Size

pathspider.chains.tcp.**TO_NOP = 1**
TCP Option - No Operation

pathspider.chains.tcp.**TO_SACK = 5**
TCP Option - Selective Acknowledgement

pathspider.chains.tcp.**TO_SACKOK = 4**
TCP Option - Selective Acknowledgement Permitted

pathspider.chains.tcp.**TO_TS = 8**
TCP Option - Timestamp

pathspider.chains.tcp.**TO_WS = 3**
TCP Option - Window Scaling

pathspider.chains.tcp.**tcp_options**(*tcp*)
Parses and extracts TCP options from a python-libtrace TCP object.

> **Warning:** This is a pure Python implementation of a TCP options parser and does not benefit from the speed advantage generally realised by calling to libtrace functions written in C through python-libtrace.

Parameters **tcp** (*plt.tcp*) – The TCP header to extract options from

Returns A mapping of option kinds to values

Return type dict

1.5.9 TCP Fast Open Chain

This module contains the TFOChain flow analysis chain which can be used by PATHspider's Observer for recording TCP Fast Open *[RFC7413]* details.

class pathspider.chains.tfo.**TFOChain**
This flow analysis chain records details of TCP Fast Open use in the flow record. It will determine whether the IANA assigned TCP option kind or the TCP Option Experiment ID *[RFC6994]* was used to identify the option, and whether the data sent on the SYN was acknowledged.

Field Name	Type	Meaning
tfo_synkind	int	Identified by pathspider.chains.tcp.TO_{FASTOPEN,EXPA,EXPB}
tfo_ackkind	int	Identified by pathspider.chains.tcp.TO_{FASTOPEN,EXPA,EXPB}
tfo_synclen	int	TFO cookie length in the forward direction
tfo_ackclen	int	TFO cookie length in the reverse direction
tfo_dlen	int	Length of SYN payload in the forward direction
tfo_ack	int	Bytes acknowledged on the SYN in the reverse direction

new_flow(*rec*, *ip*)
For a new flow, all fields will be initialised to int(0).

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always True

Return type bool

tcp(*rec*, *tcp*, *rev*)
Records TCP Fast Open details.

TCP Option Used The TCP options will be parsed for options that use either the IANA assigned TCP option number or one of the TCP Option Experiment option numbers with the TCP Option Experiment ID used by TCP Fast Open early in its standardisiation. If an option is found, the method by which it was identified will be recorded in the `tfo_synkind` field for the forward direction and `tfo_ackkind` field for the reverse direction.

TCP Fast Open Cookie Length The length of the cookies observed on TCP options will be recorded in the `tfo_synclen` field for the forward direction and `tfo_ackclen` for the reverse direction. If no Fast Open option is found, this will remain at 0 when the flow is complete.

Acknowledgement of SYN data The length of the data on the SYN in the forward direction will be recorded in the `tfo_dlen` field. The TCP sequence number for the SYN in the forward direction will be recorded in `tfo_seq` field and the TCP acknowledgement number for the SYN in the reverse direction will be recorded in the `tfo_ack` field.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the TCP segment that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns Always True

Return type bool

1.5.10 UDP Chain

This module contains the UDPChain flow analysis chain which can be used by PATHspider's Observer for recording UDP details.

class pathspider.chains.udp.**UDPChain**

Field Name	Type	Meaning
udp_zero_checksum_fwd	bool	True if the last packet in the flow in the forward direction had the UDP checksum disabled (set to zero).
udp_zero_checksum_rev	bool	True if the last packet in the flow in the reverse direction had the UDP checksum disabled (set to zero).

new_flow(*rec*, *ip*)
For a new flow, all fields will be initialised to `None`.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns Always `True`

Return type bool

udp(*rec*, *udp*, *rev*)
Records details from UDP datagram about the UDP header.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the UDP packet that was observed to be part of this flow
- **rev** (*bool*) – `True` if the packet was in the reverse direction, `False` if in the forward direction

Returns Always `True`

Return type bool

1.6 Resolving Target Lists

1.6.1 Built-in DNS Resolver

The resolver accepts input formatted as CSV in the style of the Alexa top 1 million website listing:

```
rank,domain
```

The output format is the native input format for PATHspider plugins. To get started, you can use the included example list of domains:

```
sudo pspdr measure -i eth0 --csv dnsresolv </usr/share/doc/pathspider/examples/
↪dnsresolvtest.csv
```

This built in resolver is implemented as a PATHspider measurement plugin to demonstrate the flexibility of the plugin framework. For larger campaigns you may instead wish to use the advanced DNS resolver available seperately.

1.6.2 Advanced DNS Resolver

Hellfire is a parallelised DNS resolver. It is written in Go and for the purpose of generating input lists to PATHspider, though may be useful for other applications. You will require Go to be installed on your computer before you can use Hellfire.

Installation is via `go get`:

```
go get pathspider.net/hellfire/...
```

The following input types are supported:

- Alexa Top 1 Million Global Sites
- Cisco Umbrella 1 Million
- Citizen Lab Test Lists
- OpenDNS Public Domain Lists
- Comma-Seperated Values Files
- Plain Text Domain Lists

More information on usage can be found at the Hellfire website.

1.7 Developing Plugins

PATHspider is written to be extensible and the plugins that are included in the PATHspider distribution are examples of the measurements that PATHspider can perform.

`pathspider.plugins` is a namespace package. Namespace packages are a mechanism for splitting a single Python package across multiple directories on disk. One or more distributions may provide modules which exist inside the same namespace package. The PATHspider distribution's plugins are installed in `pathspider.plugins`, but also 3rd-party plugins can exist in this path without being a part of the PATHspider distribution.

1.7.1 Choosing a Plugin Model

To be as flexible as possible while using real network stacks, PATHspider has 4 models for plugins. The following flowchart can help you to decide which model best suits your use case:

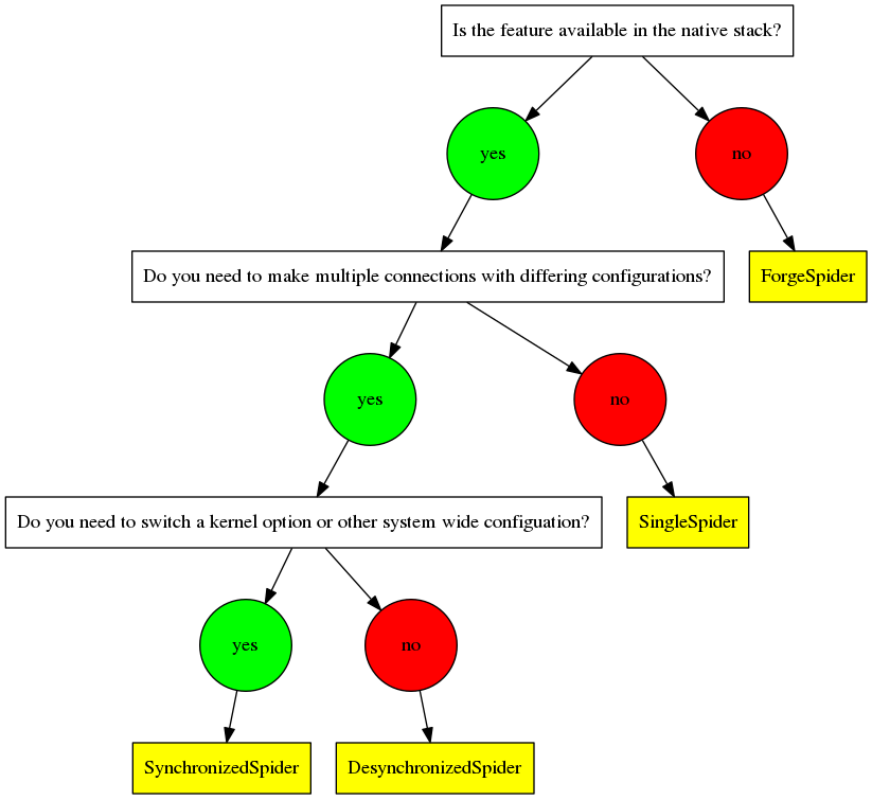

Fig. 1.2: A flowchart to help choosing a model for your plugin

SynchronizedSpider Model

This is the original model, where connection logic is built-in to PATHspider and the plugin provides functions for altering the system-wide configuration. Originally developed for testing with explicit congestion notification, this model can be used wherever iptables rules, sysctl flags or other system-wide configuration changes can be used to enable or disable a particular feature of a network protocol.

DesynchronizedSpider Model

This model was developed for TCP fast open where synchronization was not required. In this case, socket options are used to control whether or not the feature is used and so the global synchronization only serves to slow down measurements. The connection helper you are using must support the feature which means for most helpers that it must be supported by both libcurl and pycurl.

SingleSpider Model

This model was developed for TCP Maximum Segment Size discovery. In this case, only a single connection is required with no global configuration or customisation of the connection. All work is done by the Observer or using the output of the connection helpers.

ForgeSpider Model

This model was developed for evil bit testing where there was no support in the native stack. It uses Scapy to forge packets and so is the most flexible, but care must be taken to ensure that the baseline test is truly representative of existing traffic on the Internet. As an example, some firewall drop TCP packets that do not use timestamps as a "defense" against forged packets.

1.7.2 Plugin Basics

Quickstart

The directory layout and example plugin below can be found in the pathspider-example GitHub repository. You can get going quickly by forking this repository and using that as a basis for plugin development. The repository has templates for a Synchronized, a Desynchronized and a Forge plugin.

Directory Layout

`pathspider.plugins` is a namespace package. Namespace packages are a mechanism for splitting a single Python package across multiple directories on disk. One or more distributions may provide modules which exist inside the same namespace package. The PATHspider distribution's plugins are installed in `pathspider.plugins`, but also 3rd-party plugins can exist in this path without being a part of the PATHspider distribution.

To get started you will need to create the required directory layout for PATHspider plugins, in this case for the Example plugin:

```
pathspider-example
└── pathspider
    ├── __init__.py
    └── plugins
        ├── __init__.py
        └── example.py
```

Inside both __init__.py files, you will need to add the following (and only the following):

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

Your plugin will be written in `example.py` and this plugin will be discovered automatically when you run PATHspider.

Running Your Plugin

In order to run your plugin, in the root of your plugin source tree run:

```
PYTHONPATH=. pspdr measure -i eth0 example </usr/share/doc/pathspider/examples/
↪webtest.ndjson
```

Unless you install your plugin, you will need to add the plugin tree to the `PYTHONPATH` to allow the plugin to be discovered.

1.7.3 Common Plugin Features

Plugin Metadata

All plugins should contain basic metadata that is used internally within PATHspider for generating help text and command line options. This takes the form of class variables that by convention are at the start of the class.

Name	Description
name	A short name for the plugin used in the command line invocation of PATHspider
description	A human readable description of the plugin used in help text
version	A version number for the plugin

For example, from the DSCP plugin:

```
class DSCP(SynchronizedSpider, PluggableSpider):

    name = "dscp"
    description = "Differentiated Services Codepoints"
    version = "1.0.0"
```

Note: Plugins that ship with PATHspider set version to `pathspider.base.__version__`. This should only be done by plugins that are part of the PATHspider distribution as this allows these plugins to have the same version as PATHspider, which would be useless for 3rd-party plugins that release independently.

Command Line Arguments

Depending on the type of plugin, default command line arguments will be added for your plugin. You can add additional command line arguments by adding a static method to your plugin named *extra_args()*.

For example, from the DSCP plugin:

```
@staticmethod
def extra_args(parser):
    parser.add_argument(
            "--codepoint",
            type=int,
            choices=range(0, 64),
            default='48',
            metavar="[0-63]",
            help="DSCP codepoint to send (Default: 48)")
```

1.7.4 SynchronizedSpider Development

SynchronizedSpider plugins use built-in connection methods along with global system configuration to change the behaviour of the connections.

Connection Modes

The following connection types are built-in to PATHspider:

Name	Description
tcp	Perform a TCP handshake
http	Perform an HTTP GET request
https	Perform an HTTP GET request using TLS
dnsudp	Perform a DNS query using UDP
dnstcp	Perform a DNS query using TCP

To indicate the connection types that are supported by your plugin, use the `connect_supported` metadata variable. The first type listed in the variable will be the default connection type for the plugin.

For example, if your plugin supports all the TCP based connection types and you would like plain HTTP to be the default:

```
class SynchronizedSpiderPlugin(SynchronizedSpider, PluggableSpider):
    connect_supported = ["http", "https", "tcp", "dnstcp"]
```

Configuration Functions

Configuration functions are at the heart of a SynchronizedSpider plugin. These may make calls to `sysctl` or `iptables` to make changes to the way that traffic is generated.

One function should be written for each of the configurations and PATHspider will ensure that the configurations are set before the corresponding traffic is generated. It is the responsibility of plugin authors to ensure that any configuration is reset by the next configuration function if that is required.

By convention, functions should be prefixed with `config_` to ensure there are no conflicts. After declaring the functions, you must then set the `configurations` metadata variable with pointers to each of the configuration functions.

The following shows the relevant portions of the ECN plugin, which uses this framework:

```
class ECN(SynchronizedSpider, PluggableSpider):
    def config_no_ecn(self): # pylint: disable=no-self-use
        """
        Disables ECN negotiation via sysctl.
        """

        logger = logging.getLogger('ecn')
        subprocess.check_call(
            ['/sbin/sysctl', '-w', 'net.ipv4.tcp_ecn=2'],
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL)
        logger.debug("Configurator disabled ECN")

    def config_ecn(self): # pylint: disable=no-self-use
        """
```

```
        Enables ECN negotiation via sysctl.
        """

        logger = logging.getLogger('ecn')
        subprocess.check_call(
            ['/sbin/sysctl', '-w', 'net.ipv4.tcp_ecn=1'],
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL)
        logger.debug("Configurator enabled ECN")

    configurations = [config_no_ecn, config_ecn]
```

Warning: You must have the `configurations` variable *after* the declaration of the functions, as otherwise you are attempting to reference functions that have not yet been defined.

1.7.5 DesynchronizedSpider Development

DesynchronizedSpider plugins modify the connection logic in order to change the behaviour of the connections. There is no global state synchronisation and so a DesynchronizedSpider can be more efficient than a SynchronizedSpider.

Connection Functions

Connection functions are at the heart of a DesynchronizedSpider plugin. These use a connection helper (or custom connection logic) to generate traffic towards with a target to get a reply from the target.

One function should be written for each connection to be made, usually with at least two function to provide a baseline followed by an experimental connection.

By convention, functions should be prefixed with *conn_* to ensure there are no conflicts. After declaring the functions, you must then set the connections metadata variable with pointers to each of the connection functions.

The following shows the relevant portions of the H2 plugin, which uses this framework:

```
class H2(DesynchronizedSpider, PluggableSpider):
    def conn_no_h2(self, job, config):  # pylint: disable=unused-argument
        if self.args.connect == "http":
            return connect_http(self.source, job, self.args.timeout)
        if self.args.connect == "https":
            return connect_http(self.source, job, self.args.timeout)
        else:
            raise RuntimeError("Unknown connection mode specified")

    def conn_h2(self, job, config): # pylint: disable=unused-argument
        curlopts = {pycurl.HTTP_VERSION: pycurl.CURL_HTTP_VERSION_2_0}
        curlinfos = {pycurl.INFO_HTTP_VERSION}
        if self.args.connect == "http":
            return connect_http(self.source, job, self.args.timeout, curlopts,␣
↪curlinfos)
        if self.args.connect == "https":
            return connect_https(self.source, job, self.args.timeout, curlopts,␣
↪curlinfos)
        else:
            raise RuntimeError("Unknown connection mode specified")
```

```
    connections = [conn_no_h2, conn_h2]
```

1.7.6 SingleSpider Development

SingleSpider uses the built-in connection helpers to make a single connection to the target which is optionally observed by Observer chains.

This is the simplest model and only requires a `combine_flows` function to generate conditions from the connection helper output and flow record output from the Observer.

1.7.7 ForgeSpider Development

ForgeSpider plugins use Scapy to send forged packets to targets.

Plugin Metadata

As well as the common metadata, ForgeSpider plugins also require a `packets` variable, containing the number of different packets that should be generated for each target.

For example, if you had two different packets to be sent:

```
class ForgeSpiderPlugin(ForgeSpider, PluggableSpider):
    packets = 2
```

Packet Forging

As ForgeSpider uses Scapy, you will need to import any features from Scapy you wish to use in order to construct your packets. Scapy provides a flexible toolbox for packet forging, to learn more please refer to the Scapy project's documentation.

The heart of a ForgeSpider is the `forge()` function. This function takes two arguments, the job containing the target information and the sequence number. This function will be called the number of times set in the *packets* metadata variable and *seq* will be set to the number of times the function has been called for this job.

The function must return a Scapy Layer 3 packet. As a very basic example, a function that forges a TCP SYN first, then a TCP RST:

```
def forge(self, job, seq):
    sport = 0
    while sport < 1024:
        sport = int(RandShort())
    l4 = TCP(sport=sport, dport=job['dp'])
    ip = IP(src=self.source[0], dst=job['dip'])
    if seq == 0:
        l4.flags = "S"
    if seq == 1:
        l4.flags = "R"
    return ip/l4
```

As jobs may be for both IPv4 and IPv6 targets, you should account for this and build your packets using the correct Scapy functions for the IP version. ForgeSpider also supports the `--connect` option and you can use this to modify the type of packets generated in the forge function.

1.7.8 Flow Analysis Chains

PATHspider's flow observer accepts analysis chains and passes python-libtrace dissected packets along with the associated flow record to them for every packet recieved. The chains that are desired should be specified in the `chains` attribute of your class as a list of classes. If this list is empty, which is the default if not overridden in your class, no flow analysis will be performed. This can be used during early development of your plugin while you work on the traffic generation.

When you are ready to start working with flow analysis, you will need to expand your `chains` attribute. You can see this in the following example:

```
from pathspider.chains.basic import BasicChain

class Example(SynchronizedSpider, PluggableSpider):

    name = "example"
    description = "An Example Plugin"
    version = "1.0"
    chains = [BasicChain, ...]

    ...
```

Depending on the types of analysis you would like to do on the packets, you should add additional chains to the chains attribute of your plugin class.

Library Flow Analysis Chains

The *pathspider.chains.basic.BasicChain* chain creates inital state for the flow record, extracting the 5-tuple and counting the number of packets and octets in each direction. Unless you have good reason, this chain should be included in your plugin as its fields are used by the merger to match flow records with their corresponding jobs.

PATHspider also provides library flow analysis chains for some protocols and extensions which you can find in the *Observer* page.

Writing Flow Analysis Chains

When you are ready to write a chain for the observer, first identify which data should be stored in the flow record. This is a `dict` that is made available for every call to a chain function for a particular flow (identified by its 5-tuple) and not shared across flows.

Flow chains inherit from `pathspider.chains.base.Chain` and provide a series of functions for handling different types of packet.

class pathspider.chains.base.**Chain**
: This is an abstract flow analysis chain. It is intended that all flow analysis chains will subclass this class and it is not intended for this class to be directly used by PATHspider plugins.

You should familiarise yourself with the python-libtrace documentation. The analysis functions all follow similar function prototypes with `rec`: the flow record, `x`: the protocol header, and `rev`: boolean value indicating the direction the packet travelled (i.e. was the packet in the reverse direction?). The exception to this rule is for `icmp4` and `icmp6` which also provide a `q` argument, the ICMP quotation if the message was a type that carries a quotation otherwise this is set to `None`.

The only requirement for a flow analysis chain is that it provides a `new_flow()` function. All other functions are optional. If the `new_flow()` function does not return True, the flow will be discarded. All other functions must return `True` unless they have identified that the flow is complete and should be passed on to the merger. If this is not easily detectable, a timeout will pass the flow for merging after a fixed interval where no new packets have been seen.

You can find descriptions for each of the possible chain functions in `pathspider.chains.noop.NoOpChain`:

class `pathspider.chains.noop.`**NoOpChain**

This flow analysis chain does not perform any analysis and is present here for the purpose of documentation and testing.

icmp4(*rec*, *ip*, *q*, *rev*)

This function is called for every new ICMPv4 packet seen. It can be used to record details for fields present in the ICMPv4 header or quotation.

Note: The IP header is passed as the argument, not the ICMP header as it may be desirable to access fields in the IP header, for instance to determine the router or host that sent the ICMP message

---

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow and contained an ICMPv4 header
- **q** (*plt.ip*) – the ICMP quotation of the packet that triggered this message (if any)
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

icmp6(*rec*, *ip6*, *q*, *rev*)

This function is called for every new ICMPv6 packet seen. It can be used to record details for fields present in the ICMPv6 header or quotation.

Note: The IP header is passed as the argument, not the ICMP header as it may be desirable to access fields in the IP header, for instance to determine the router or host that sent the ICMP message

---

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip6*) – the IPv6 packet that was observed to be part of this flow and contained an ICMPv6 header
- **q** (*plt.ip6*) – the ICMP quotation of the packet that triggered this message (if any)
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

ip4(*rec*, *ip*, *rev*)

This function is called for every new IPv4 packet seen. It can be used to record details for fields present in the IPv4 header.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip*) – the IPv4 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

ip6(*rec*, *ip6*, *rev*)

This function is called for every new IPv6 packet seen. It can be used to record details for fields present in the IPv6 header.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip6*) – the IPv6 packet that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

new_flow(*rec*, *ip*)

This function is called for every new flow to initialise a flow record with the fields that will be used by this chain. It is recommended to initialise all fields to None until other functions have set values for them to make clear which fields are set by this chain and to avoid key errors later.

Parameters

- **rec** (*dict*) – the flow record
- **ip** (*plt.ip or plt.ip6*) – the IP or IPv6 packet that triggered the creation of a new flow record

Returns True if flow should be kept, False if flow should be discarded

Return type bool

tcp(*rec*, *tcp*, *rev*)

This function is called for every new TCP packet seen. It can be used to record details for fields present in the TCP header.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the TCP segment that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

udp (*rec*, *udp*, *rev*)
This function is called for every new UDP packet seen. It can be used to record details for fields present in the UDP header.

Parameters

- **rec** (*dict*) – the flow record
- **tcp** – the UDP segment that was observed to be part of this flow
- **rev** (*bool*) – True if the packet was in the reverse direction, False if in the forward direction

Returns True if flow should continue to be observed, False if the flow should be passed on for merging (i.e. the flow is complete)

Return type bool

1.8 PATHspider Internals

To learn more about the internals of PATHspider, you can find here an overview of the key classes that make up the individual parts of the architecture:

1.8.1 Abstract Spider

The core functionality of PATHspider plugins is implemented in two classes: *pathspider.sync.SynchronizedSpider* and *pathspider.desync.DesynchronizedSpider*. There is also a third class, *pathspider.forge.ForgeSpider* that inherits from *pathspider.desync.DesynchronizedSpider*. These both inherit from the base *pathspider.base.Spider* which provides a skeleton that has the required functions for any plugin. The documentation for this base class is below:

pathspider.base

Basic framework for Pathspider: coordinate active measurements on large target lists with both system-level network stack state (sysctls, iptables rules, etc) as well as information derived from flow-level passive observation of traffic at the sender.

class pathspider.base.**PluggableSpider**

static register_args(*subparsers*)

class pathspider.base.**Spider**(*worker_count*, *libtrace_uri*, *args*, *server_mode*)
A spider consists of a configurator (which alternates between two system configurations), a large number of workers (for performing some network action for each configuration), an Observer which derives information from passively observed traffic, and a thread that merges results from the workers with flow records from the collector.

add_job(*job*)
Adds a job to the job queue.

If PATHspider is currently stopping, the job will not be added to the queue.

chains = []

combine_flows(*flows*)

configurator()

create_observer()
Create a flow observer.

This function is called by the base Spider logic to get an instance of *pathspider.observer.Observer* configured with the function chains that are requried by the plugin.

exception_wrapper(*target*, **args*, ***kwargs*)

merge(*flow*, *res*)
Merge a job record with a flow record.

Parameters

- **flow** (*dict*) – The flow record.
- **res** (*dict*) – The job record.

Returns tuple – Final record for job.

In order to create a final record for reporting on a job, the final job record must be merged with the flow record. This function should be implemented by any plugin to provide the logic for this merge as the keys used in these records cannot be known by PATHspider in advance.

This method is not implemented in the abstract *pathspider.base.Spider* class and must be implemented by any plugin.

merger()
Thread to merge results from the workers and the observer.

post_connect(*job*, *rec*, *config*)
Performs post-connection operations.

Parameters

- **job** (*dict*) – The job record.
- **rec** (*dict*) – The result of the connection operation(s).
- **config** (*int*) – The state of the configurator during `pathspider.base.Spider.connect()`.

The post_connect function can be used to perform any operations that must be performed after each connection. It will be run for both the A and the B configuration, and is not synchronized with the configurator.

Plugins to PATHspider can optionally implement this function. If this function is not overloaded, it will be a noop.

Any sockets or other file handles that were opened during `pathspider.base.Spider.connect()` should be closed in this function if they have not been already.

pre_connect(*job*)
Performs pre-connection operations.

Parameters **job** (*dict*) – The job record

The pre_connect function can be used to perform any operations that must be performed before each connection. It will be run only once per job, with the same result passed to both the A and B connect calls. This function is not synchronized with the configurator.

Plugins to PATHspider can optionally implement this function. If this function is not overloaded, it will be a noop.

shutdown()
Shut down PathSpider in an orderly fashion, ensuring that all queued jobs complete, and all available results are merged.

start()
This function starts a PATHspider plugin by:

- Setting the running flag
- Create and start an observer
- Start the merger thread
- Start the configurator thread
- Start the worker threads

The number of worker threads to start was given when activating the plugin.

terminate()
Shut down PathSpider as quickly as possible, without any regard to completeness of results.

worker(*worker_number*)

1.8.2 Desynchronized Spider

This abstract class can be extended to produce new plugins that do not require a system-wide configuration synchronization.

pathspider.desync

class pathspider.desync.**DesynchronizedSpider**(*worker_count*, *libtrace_uri*, *args*, *server_mode=False*)

configurator()
Since there is no need for a configurator thread in a desynchronized spider, this thread is a no-op

connections = []

classmethod register_args(*subparsers*)

worker(*worker_number*)
This function provides the logic for configuration-synchronized worker threads.

Parameters **worker_number** (*int*) – The unique number of the worker.

The workers operate as continuous loops:

- Fetch next job from the job queue
- Perform pre-connection operations
- Acquire a lock for "config_zero"
- Perform the "config_zero" connection
- Release "config_zero"
- Acquire a lock for "config_one"
- Perform the "config_one" connection
- Release "config_one"
- Perform post-connection operations for config_zero and pass the result to the merger
- Perform post-connection operations for config_one and pass the result to the merger
- Do it all again

If the job fetched is the SHUTDOWN_SENTINEL, then the worker will terminate as this indicates that all the jobs have now been processed.

1.8.3 Forge Spider

This abstract class can be extended to produce new plugins that are using Scapy for packet forging.

pathspider.forge

class pathspider.forge.**ForgeSpider**(*worker_count*, *libtrace_uri*, *args*)

chains = [<class 'pathspider.chains.basic.BasicChain'>]

connect(*job*, *seq*)

forge(*job*, *config*)

packets = 0

pre_connect(*job*)

classmethod register_args(*subparsers*)

setup(*job*)

1.8.4 Observer

class pathspider.observer.**DummyObserver**

The dummy observer provides a class compatible with the API of the Observer class without actually performing any operations. This is primarily used for PATHspider's test suite.

run_flow_enqueuer(*flowqueue*, *irqueue=None*)

When running the flow enqueuer, no network operation is performed and the thread will block until given a shutdown signal. When the shutdown signal is received it will cascade the signal onto the flowqueue in the same way that a real Observer instance would.

class pathspider.observer.**Observer**(*lturi*, *chains=None*, *idle_timeout=30*, *expiry_timeout=5*)

Wraps a packet source identified by a libtrace URI, parses packets to divide them into flows, passing these packets and flows onto a function chain to allow data to be associated with each flow.

__init__(*lturi*, *chains=None*, *idle_timeout=30*, *expiry_timeout=5*)

Create an Observer.

Parameters **chains** – Array of Observer chain classes

See also *Observer Documentation*

flush()

run_flow_enqueuer(*flowqueue*, *irqueue=None*)

class pathspider.observer.**PacketClockTimer**(*time*, *fn*)

fn

Alias for field number 1

time

Alias for field number 0

1.8.5 Synchronized Spider

This abstract class can be extended to produce new plugins that require a system-wide configuration synchronization.

pathspider.sync

class pathspider.sync.**SemaphoreN**(*value*)
An extension to the standard library's BoundedSemaphore that provides functions to handle n tokens at once.

acquire_n(*value=1*, *blocking=True*, *timeout=None*)
Acquire `value` number of tokens at once.

The parameters `blocking` and `timeout` have the same semantics as `BoundedSemaphore`.

Returns The same value as the last call to *BoundedSemaphore*'s `acquire()` if `acquire()` were called `value` times instead of the call to this method.

empty()
Acquire all tokens of the semaphore.

release_n(*value=1*)
Release `value` number of tokens at once.

Returns The same value as the last call to *BoundedSemaphore*'s `release()` if `release()` were called `value` times instead of the call to this method.

class pathspider.sync.**SynchronizedSpider**(*worker_count*, *libtrace_uri*, *args*, *server_mode=False*)

configurations = []

configurator()
Thread which synchronizes on a set of semaphores and alternates between two system states.

connect(*job*, *config*)
Performs the requested connection.

classmethod register_args(*subparsers*)

worker(*worker_number*)
This function provides the logic for configuration-synchronized worker threads.

Parameters **worker_number** (*int*) – The unique number of the worker.

The workers operate as continuous loops:

- Fetch next job from the job queue
- Perform pre-connection operations
- Acquire a lock for "config_zero"
- Perform the "config_zero" connection
- Release "config_zero"
- Acquire a lock for "config_one"
- Perform the "config_one" connection
- Release "config_one"
- Perform post-connection operations for config_zero and pass the result to the merger
- Perform post-connection operations for config_one and pass the result to the merger

- Do it all again

If the job fetched is the SHUTDOWN_SENTINEL, then the worker will terminate as this indicates that all the jobs have now been processed.

1.9 References

CHAPTER 2

Citing PATHspider

When presenting work that uses PATHspider, we would appreciate it if you could cite PATHspider as:

> Learmonth, I.R., Trammell, B., Kuhlewind, M. and Fairhurst, G., 2016, July. PATHspider: A tool for active measurement of path transparency. In Proceedings of the 2016 Applied Networking Research Workshop (pp. 62-64). ACM.

CHAPTER 3

Acknowledgements

Current development of PATHspider is supported by the European Union's Horizon 2020 project MAMI. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688421. The opinions expressed and arguments employed reflect only the authors' view. The European Commission is not responsible for any use that may be made of that information.

Bibliography

[Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M. and Tokuda, H., 11, November. Is it still possible to extend TCP?. In Proceedings of the 11 ACM SIGCOMM conference on Internet measurement conference (pp. 181-194). ACM.

[Trammell15] Trammell, B., Kühlewind, M., Boppart, D., Learmonth, I., Fairhurst, G. and Scheffenegger, R., 15, March. Enabling Internet-wide deployment of explicit congestion notification. In International Conference on Passive and Active Network Measurement (pp. 193-205). Springer International Publishing.

[Gubser15] Gubser, E., Measuring Explicit Congestion Negotiation (ECN) support based on P2P networks, 2015.

[RFC768] Postel, J., 1980, **User Datagram Protocol**. RFC Editor.

[RFC792] Postel, J., 1981, **Internet Control Message Protocol**. RFC Editor.

[RFC793] Postel, J., 1981, **Transmission Control Protocol**. RFC Editor.

[RFC1035] Mockapetris, P., 1987. **Domain Names - Impelmentation and Specification**. RFC Editor.

[RFC2474] Nichols, K., Blake, S., Baker, F. and Black, D., 1998. **Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers**. RFC Editor.

[RFC3168] Ramakrishnan, K., Floyd, S. and Black, D., 2001. **The addition of explicit congestion notification (ECN) to IP**. RFC Editor.

[RFC3260] Grossman, D., 2002. **New terminology and clarifications for DiffServ**. RFC Editor.

[RFC4443] Conta, A., Deering, S., Gupta, M., 2006. **Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification**. RFC Editor.

[RFC6994] Touch, J., 2013. **Shared Use of Experimental TCP Options**. RFC Editor.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S. and Jain, A., 2014. **TCP Fast Open**. RFC Editor.

[RIPEAtlas] Ripe, N.C.C.. RIPE atlas.

[Filasto12] Filasto, A. and Appelbaum, J., 2012, August. OONI: Open Observatory of Network Interference. In FOCI.

[Kreibich10] Kreibich, C., Weaver, N., Nechaev, B. and Paxson, V., 2010, November. Netalyzr: illuminating the edge network. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (pp. 246-259). ACM.

Python Module Index

p

Index

O

P

R

S

T

U

W