Trixy Documentation

Release 2.0.0b0pre

Austin Hartzheim

April 22, 2015

Contents

CHAPTER 1

Getting Started

Here are some instructions for getting started with Trixy.

1.1 Installing Trixy

Installing Trixy is just a simple command. Note that you should use the Python 3 version:

```
sudo pip install trixy
```

Alternatively, you can download a source tarball or zip file from PyPI or Github. Then, you can extract it and install it by running:

```
sudo python3 setup.py install
```

1.2 The Basics: Theory

Trixy is structured into four component classes: servers, inputs, outputs, and processors. Servers are responsible for capturing incoming connections and passing them to an input class. The input class then takes these connections and builds processing chains for them. These processing chains consist of processors, which modify data passing through them, and outputs, which forward the data stream (including any modifications) to a remote host.

To use Trixy, you should import it into your Python project and create subclasses of `trixy.TrixyInput`. Inside the `__init__()` method of the subclass, you should create a chain of nodes which the data should pass through. As an example:

```
processor = trixy.TrixyProcessor()
self.connect_node(processor)
processor.connect_node(trixy.TrixyOutput('127.0.0.1', 9999))
```

The first line creates a processor node. The default `trixy.TrixyProcessor` class does not do anything other than forward the data, so you should create a subclass and override some of its methods to modify its behavior (covered next). The second line connects the input instance with this processor node so that the input will forward the data it gets to the processor. The last line connects the processor node to a `trixy.TrixyOutput` instance that is created at the same time. This causes the processor to forward data it gets to the output (after making any modifications). The default output that is used in this case creates a TCP connection to localhost on port 9999 and forwards the data there.

1.3 Modifying Data: Custom Processors

Trixy is great for simply re-routing data, but its realy power lies in its ability to process the data on the fly. To do this, you need to create a custom `trixy.TrixyProcessor` subclass.

When you are creating your own custom processor, you should modify packets like so:

```
class CustomProcessor(trixy.TrixyProcessor):
    def handle_packet_down(self, data):
        # Modify the data variable here
        self.forward_packet_down(data)

    def handle_packet_up(self, data):
        # Modify the data variable here
        self.forward_packet_up(data)
```

The `handle_packet_down()` method is called to process data flowing from the input to the output. The `handle_packet_up()` method is used to process data moving from the output to the input. The calls to the `forward_packet_down()` and `forward_packet_up()` then send the modified data on its way to the next node(s) in the chain.

Note: It is also the case that you can ommit calls to `forward_packet_down()` and `forward_packet_up()` when you want to drop a packet.

CHAPTER 2

Examples

Here are some examples of how to use Trixy:

2.1 Passthrough Proxy

The following code creates a Trixy proxy server on a local port and then sends the output to austinhartzheim.me on port 80:

```
# /usr/bin/env python3
import asyncore
import trixy

class CustomInput(trixy.TrixyInput):
    def __init__(self, sock, addr):
        super().__init__(sock, addr)

        # This output class connects to this hostname/port by default
        output = trixy.TrixyOutput('austinhartzheim.me', 80)
        self.connect_node(output)

if __name__ == '__main__':
    # Run the Trixy server on localhost, port 8080
    server = trixy.TrixyServer(CustomInput, '127.0.0.1', 8080)
    asyncore.loop()
```

This example was taken from the README file.

2.2 Changing Website Responses

The following example takes an incoming connection on a local port, redirects it to a remove webserver on port 80 (specifically, the example.com server), and then modifies the response from example.com:

```
#! /usr/bin/env python3
import asyncore
import trixy

REMOTE_ADDR = '93.184.216.119' # IP for example.com
REMOTE_PORT = 80
```

```
class ExampleReplacer(trixy.TrixyProcessor):

    def handle_packet_up(self, data):
        data = data.replace(b'Example Domain', b'Win Domain!')
        self.forward_packet_up(data)


class CustomInput(trixy.TrixyInput):
    def __init__(self, sock, addr):
        super().__init__(sock, addr)

        processor = ExampleReplacer()
        self.connect_node(processor)

        output = trixy.TrixyOutput(REMOTE_ADDR, REMOTE_PORT)
        processor.connect_node(output)
        print(processor.upstream_nodes)


if __name__ == '__main__':
    server = trixy.TrixyServer(CustomInput, '0.0.0.0', 80)
    asyncore.loop()
```

This example was originally posted on the developer's website.

2.3 More Eamples Soon

More examples are on their way! But, if you write one first, feel free to send a pull request on Github.

CHAPTER 3

Trixy Code Documentation

3.1 trixy

The main Trixy module contains the parent classes that can be modified for custom functionality.

class trixy.**TrixyInput**(*sock*, *addr*)

Bases: trixy.TrixyNode, asyncore.dispatcher_with_send

Once a connection is open, establish an output chain.

add_downstream_node(*node*)

Add a one direction downstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a unidirectional link to.

add_upstream_node(*node*)

Add a one direction upstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The upstream node to create a unidirectional link to.

connect_node(*node*)

Create a bidirectional connection between the two nodes with the downstream node being the parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a bidirectional connection to.

forward_packet_down(*data*)

Forward data to all downstream nodes.

Parameters **data** (*bytes*) – The data to forward.

forward_packet_up(*data*)

Forward data to all upstream nodes.

Parameters **data** (*bytes*) – The data to forward.

handle_packet_down(*data*)

Hadle data moving downwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

class trixy.**TrixyNode**

Bases: builtins.object

A base class for TrixyNodes that implements some default packet forwarding and node linking.

add_downstream_node(*node*)
Add a one direction downstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a unidirectional link to.

add_upstream_node(*node*)
Add a one direction upstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The upstream node to create a unidirectional link to.

connect_node(*node*)
Create a bidirectional connection between the two nodes with the downstream node being the parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a bidirectional connection to.

forward_packet_down(*data*)
Forward data to all downstream nodes.

Parameters **data** (*bytes*) – The data to forward.

forward_packet_up(*data*)
Forward data to all upstream nodes.

Parameters **data** (*bytes*) – The data to forward.

handle_close(*direction='down'*)
The connection has closed on one end. So, shutdown what we are doing and notify the nodes we are connected to.

Parameters **direction** (*str*) – 'down' or 'up' depending on if downstream nodes need to be closed, or upstream nodes need to be closed.

handle_packet_down(*data*)
Hadle data moving downwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

handle_packet_up(*data*)
Hadle data moving upwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

class trixy.**TrixyOutput**(*host*, *port*, *autoconnect=True*)
Bases: trixy.TrixyNode, asyncore.dispatcher_with_send

Output the data, generally to another network service.

add_downstream_node(*node*)
Add a one direction downstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a unidirectional link to.

add_upstream_node(*node*)
Add a one direction upstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The upstream node to create a unidirectional link to.

assume_connected(*host*, *port*, *sock*)
Assume that the connection has already been made. Setup all state accordingly. This is useful in situations where one output wants to pass off work to a different output (for example, a proxy output might establish the connection and then pass it off to an SSL output (which needs to act on the raw socket object).

connect_node(*node*)
Create a bidirectional connection between the two nodes with the downstream node being the parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a bidirectional connection to.

forward_packet_down(*data*)
Forward data to all downstream nodes.

Parameters **data** (*bytes*) – The data to forward.

forward_packet_up(*data*)
Forward data to all upstream nodes.

Parameters **data** (*bytes*) – The data to forward.

handle_packet_up(*data*)
Hadle data moving upwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

setup_socket(*host*, *port*, *autoconnect=True*)
Establish the outbound connection.

Parameters

- **host** (*str*) – The hostname to connect to.
- **port** (*int*) – The port on the host to connect to.
- **autoconnect** (*bool*) – Should the connection be established now, or should it be manually triggered later?

supports_assumed_connections = True
Denotes whether assumed connections are assumed by the class.

class trixy.**TrixyProcessor**
Bases: trixy.TrixyNode

Perform processing on data moving through Trixy.

add_downstream_node(*node*)
Add a one direction downstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a unidirectional link to.

add_upstream_node(*node*)
Add a one direction upstream link to the node parameter.

Parameters **node** (*TrixyNode*) – The upstream node to create a unidirectional link to.

connect_node(*node*)
Create a bidirectional connection between the two nodes with the downstream node being the parameter.

Parameters **node** (*TrixyNode*) – The downstream node to create a bidirectional connection to.

forward_packet_down(*data*)
Forward data to all downstream nodes.

Parameters **data** (*bytes*) – The data to forward.

forward_packet_up(*data*)
Forward data to all upstream nodes.

Parameters **data** (*bytes*) – The data to forward.

handle_close(*direction='down'*)
The connection has closed on one end. So, shutdown what we are doing and notify the nodes we are connected to.

Parameters **direction** (*str*) – 'down' or 'up' depending on if downstream nodes need to be closed, or upstream nodes need to be closed.

handle_packet_down(*data*)
Hadle data moving downwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

handle_packet_up(*data*)
Hadle data moving upwards. TrixyProcessor children should perform some action on *data* whereas *TrixyOutput* children should send the data to the desired output location.

Generally, the a child implementation of this method should be implemented such that it calls self.forward_packet_down with the data (post-modification if necessary) to forward the data to other processors in the chain. However, if the processor is a filter, it may drop the packet by omitting that call.

Parameters **data** (*bytes*) – The data that is being handled.

class trixy.**TrixyServer**(*tinput*, *host*, *port*)
Bases: asyncore.dispatcher

Main server to grab incoming connections and forward them.

3.2 trixy.encryption

The Trixy encryption module holds inputs and outputs that have support for encryption that applications might expect. For example, the trixy.encryption.TrixySSLInput can be used to trick a browser into thinking it is creating an encrypted connection, but the connection can then be re-routed through an unencrypted trixy.TrixyOutput for easier monitoring.

class trixy.encryption.**TrixySSLInput**(*sock*, *addr*, ***kwargs*)
Acts like a normal TrixyInput, but uses Python's ssl.wrap_socket() code to speak the SSL protocol back to applications that expect it.

class trixy.encryption.**TrixySSLOutput**(*host*, *port*, *autoconnect=True*, ***kwargs*)
Acts like a normal TriyOutput, but uses Python's ssl.wrap_socket() code to speak the SSL protocol to servers that expect it.

By default this class allows for SSL2 and SSL3 connections in addition to TLS. If you want to specify different settings, you can pass your own context to setup_socket().

assume_connected(*host*, *port*, *sock*, *context=None*, ***kwargs*)
Assume a connection that is already in progress and encrypt the traffic with a default or provded SSL context.

Parameters

- **host** (*str*) – The hostname the output should connect to.
- **port** (*int*) – The port this output should connect to.
- **sock** (*socket.socket*) – The connected socket object.
- **context** (*ssl.SSLContext*) – this optional parameter allows for custom security settings such as certificate verification and alternate SSL/TLS versions support.
- ****kwargs** – Anything else that should be passed to the SSLContext's wrap_socket method.

setup_socket(*host*, *port*, *autoconnect*, *context=None*, ***kwargs*)

Parameters

- **host** (*str*) – The hostname the output should connect to.
- **port** (*int*) – The port this output should connect to.
- **autoconnect** (*bool*) – Should the connection be established when the __init__ method is called?
- **context** (*ssl.SSLContext*) – this optional parameter allows for custom security settings such as certificate verification and alternate SSL/TLS versions support.
- ****kwargs** – Anything else that should be passed to the SSLContext's wrap_socket method.

class trixy.encryption.**TrixyTLSOutput**(*host*, *port*, *autoconnect=True*)
Acts identical to a TrixySSLOutput, but defaults to only accepting TLS for security reasons. This makes it slightly easier to prevent downgrade attacks, especially when doing hasty testing rather than full development.

3.3 trixy.proxy

The Trixy proxy inputs speak a variety of common proxy protocols, such as SOCKS4, SOCKS4a, and SOCKS5. Their default behavior is to act as a normal proxy and open a connection to the desired endpoint. However, this behavior can be overridden to create different results.

Additionally, the proxy outputs allow a connection to be subsequently made to a proxy server. This allows intercepted traffic to be easily routed on networks that require a proxy. It also makes it easier to route traffic into the Tor network.

class trixy.proxy.**Socks4Input**(*sock*, *addr*)
Implements the SOCKS4 protocol as defined in this document: http://www.openssh.com/txt/socks4.protocol

handle_connect_request(*addr*, *port*, *userid*)
The application connecting to this SOCKS4 input has requested that a connection be made to a remote host. At this point, that request can be accepted, modified, or declined.

The default behavior is to accept the request as-is.

handle_proxy_request(*data*)
In SOCKS4, the first packet in a connection is a request to either initiate a connection to a remote host and port, or it is a request to bind a port. This method is responsible for processing those requests.

reply_request_failed(*addr*, *port*)
Send a reply stating that the request was rejected (perhaps due to a firewall rule forbidding the connection or binding) or that it failed (i.e., the remote host could not be connected to or the requested port could not be bound).

reply_request_granted(*addr*, *port*)
Send a reply stating that the connection or bind request has been granted and that the connection or bind attempt was successfully completed.

reply_request_rejected(*addr*, *port*)
Send a reply saying that the request was rejected because the SOCKS server could not connect to the client's identd server.

reply_request_rejected_id_mismatch(*addr*, *port*)
Send a reply saying that the request was rejected because the SOCKS server was sent an ID by the client that did not match the ID returned by identd on the client's computer.

class trixy.proxy.**Socks4aInput**(*sock*, *addr*)
Implements the SOCKS4a protocol, which is the same as the SOCKS4 protocol except for the addition of DNS resolution as described here: http://www.openssh.com/txt/socks4a.protocol

handle_connect_request(*addr*, *port*, *userid*)
The application connecting to this SOCKS4 input has requested that a connection be made to a remote host. At this point, that request can be accepted, modified, or declined.

The default behavior is to accept the request as-is.

handle_proxy_request(*data*)
In SOCKS4, the first packet in a connection is a request to either initiate a connection to a remote host and port, or it is a request to bind a port. This method is responsible for processing those requests.

class trixy.proxy.**Socks5Input**(*sock*, *addr*)
Implements the SOCKS5 protocol as defined in RFC1928. At present, only CONNECT requests are supported.

handle_connect_request(*addr*, *port*, *addrtype*)
The application connecting to this SOCKS4 input has requested that a connection be made to a remote host. At this point, that request can be accepted, modified, or declined.

The default behavior is to accept the request as-is.

handle_method_select(*methods*)
Select the preferred authentication method from the list of client-supplied supported methods. The byte object of length one should be sent to self.reply_method to notify the client of the method selection.

reply_method(*method*)
Send a reply to the user letting them know which authentication method the server has selected. If the method 0xff is selected, close the connection because no method is supported.

reply_request_granted(*addr*, *port*, *addrtype*)
Send a reply stating that the connection or bind request has been granted and that the connection or bind attempt was successfully completed.

class trixy.proxy.**Socks5Output**(*host*, *port*, *autoconnect=True*, *proxyhost='127.0.0.1'*, *proxyport=1080*)
Implements the SOCKS5 protocol as defined in RFC1928.

handle_state_change(*oldstate*, *newstate*)
Be able to process events when they occur. It allows easier detection of when events occur if it is desired to implement different responses. It also allows detection of when the proxy is ready for use and can be used to use assume_connectecd to transfer control to a TrixyOutput.

Parameters

- **oldstate** (*int*) – The old state number.
- **newstate** (*int*) – The new state number.

exception `trixy.proxy.`**`SocksProtocolError`**

Someone sent some invalid data on the wire, and this is how to deal with it.

CHAPTER 4

What is Trixy?

Trixy is designed to be used in a variety of situations involving network traffic interception, injection, and modification. The software allows you to easily get your code running between two endpoints of a network connection. This allows you to easily:

- Log protocols for reverse engineering.
- Modify packets on bidirectional connections.
- Inject traffic into a network connection.
- Develop and test protocol parsers.
- Monitor applications for suspicious network activity.
- Sanitize traffic, removing any undesired information.

Here are some practical examples of the above:

- Cheating at video games:
  - Exploit server-client trust by modifying packets indicating how much money a player has.
  - Drop packets that indicate damage to a player.
- Removing advertising and trackers from webpages.
- Performing man-in-the-middle attacks.

CHAPTER 5

Other Documentation

If you are stuck, you should also check the following sources for information about Trixy:

- The developer's website
- The Github repository

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

t

A

C

F

H

R

S

T