Dshell Developer Guide

by Joshua Edwards and Daniel E Krych
NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
Dshell Developer Guide

Joshua Edwards
ICF

Daniel E Krych
DEVCOM Army Research Laboratory

Approved for public release: distribution unlimited.
# REPORT DOCUMENTATION PAGE

1. **REPORT DATE**
   - April 2023

2. **REPORT TYPE**
   - Special Report

3. **DATES COVERED**
   - **START DATE**: February 2023
   - **END DATE**: March 2023

4. **TITLE AND SUBTITLE**
   - Dshell Developer Guide

5. **AUTHOR(S)**
   - Joshua Edwards and Daniel E Krych

6. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
   - DEVCOM Army Research Laboratory
   - ATTN: FCDD-RLA-ND
   - Aberdeen Proving Ground, MD 21005

7. **ABSTRACT**
   This report is a guide to plugin development for the decoder-shell (Dshell) framework. It provides basic examples, core function and class definitions, and an overview of data flow. This guide will help end users develop new, custom plugins as well as modify existing plugins. Dshell is an open-source, Python-based, network forensic analysis framework developed by the US Army Combat Capabilities Development Command Army Research Laboratory. It is a modular and flexible framework, which includes over 40 plugins for the analysis and decoding of network traffic using a variety of network protocols. Dshell plugins are designed to aid in the understanding of network traffic and present results to the user in a concise, useful manner via command-line interface. Dshell is a tool for network forensic analysis that can be used out of the box for simple and advanced analyses, or customized to fit an end-user’s needs. Custom Dshell plugins can be developed to parse and analyze unique network traffic protocols and data, such as malware. Existing plugins can be modified to extract different information from the protocols they currently parse, customize the programmatic actions performed on the data, or alter the outputted information when using the plugin. The Dshell GitHub repository contains the current Python 3 version as well as an archived Python 2 version available as a tarball. This developer guide only applies to the current version.

8. **SPONSOR/MONITOR’S ACRONYM(S)**
   - ARL-SR-0471

9. **DISTRIBUTION/AVAILABILITY STATEMENT**
   - Approved for public release: distribution unlimited.

10. **SUBJECT TERMS**
    - Network, Cyber, and Computational Sciences; Dshell; developer; analyst; guide; manual; network; cyber; forensics; traffic analysis; decode; pcap; monitor; dissection

11. **SECURITY CLASSIFICATION OF:**
    - **a. REPORT**: UNCLASSIFIED
    - **b. ABSTRACT**: UNCLASSIFIED
    - **C. THIS PAGE**: UNCLASSIFIED

12. **NUMBER OF PAGES**: 39

13. **NAME OF RESPONSIBLE PERSON**
    - Daniel E Krych

14. **PHONE NUMBER**
    - (410) 278-5266

---

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18
# Contents

List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

1. **Introduction**
   1

2. **Important Concepts**
   1
   2.1 Data Sources
   2
   2.2 Plugin Chain and Produce/Consume Model
   2
   2.3 Parallelization
   3

3. **Core Object Classes**
   3
   3.1 Packet
   3
   3.2 Connection
   5
   3.2.1 Server/Client Versus Destination/Source
   6
   3.2.2 Blobs
   6
   3.3 Blob
   6

4. **Example Plugin – netflow.py**
   8

5. **Plugin Types**
   9
   5.1 PacketPlugin
   9
   5.1.1 Placeholder Functions
   10
   5.1.2 Other Functions
   11
   5.2 ConnectionPlugin
   12
   Placeholder Functions
   12
   5.3 DNSPlugin
   13
   Placeholder Functions
   13
   5.4 HTTPPlugin
   14
   5.4.1 Placeholder Functions
   14
   5.4.2 Custom Classes: HTTPRequest and HTTPResponse
   14

6. **Building a Plugin**
   15
   6.1 Building an Example Plugin
   16
   6.1.1 Decide Purpose and Metadata
   16
   6.1.2 Pick a Parent Plugin
   17
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Dshell packet object class attributes with descriptions, defined in dshell/core.py</td>
<td>4</td>
</tr>
<tr>
<td>Table 2</td>
<td>Dshell connection object class attributes with descriptions, defined in dshell/core.py</td>
<td>5</td>
</tr>
<tr>
<td>Table 3</td>
<td>Dshell Blob object class attributes with descriptions, defined in dshell/core.py</td>
<td>7</td>
</tr>
<tr>
<td>Table 4</td>
<td>Dshell PacketPlugin public class attributes with descriptions, defined in dshell/core.py</td>
<td>9</td>
</tr>
<tr>
<td>Table 5</td>
<td>Dshell PacketPlugin public class attributes with descriptions, defined in dshell/core.py</td>
<td>10</td>
</tr>
<tr>
<td>Table 6</td>
<td>Dshell ConnectionPlugin public class attributes with descriptions, defined in dshell/core.py</td>
<td>12</td>
</tr>
<tr>
<td>Table 7</td>
<td>Dshell HTTPRequest class attributes with descriptions, defined in dshell/plugins/httpplugin.py</td>
<td>15</td>
</tr>
<tr>
<td>Table 8</td>
<td>Dshell HTTPResponse class attributes with descriptions, defined in dshell/plugins/httpplugin.py</td>
<td>15</td>
</tr>
<tr>
<td>Table 9</td>
<td>Overview of steps to build a Dshell plugin</td>
<td>16</td>
</tr>
<tr>
<td>Table 10</td>
<td>“Example” plugin metadata definitions</td>
<td>17</td>
</tr>
</tbody>
</table>
1. Introduction

This report is a guide to plugin development for the decoder-shell (Dshell) framework.\(^1\) It provides basic examples, core function and class definitions, and an overview of data flow. This guide will help end users develop new, custom plugins as well as modify existing plugins.

Dshell\(^1\) is an open-source, Python-based, network forensic analysis framework developed by the US Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory (ARL). It is a modular and flexible framework, which includes over 40 plugins for the analysis and decoding of network traffic using a variety of network protocols. Dshell plugins are designed to aid in the understanding of network traffic and present results to the user in a concise, useful manner via command-line interface (CLI).

Dshell\(^1\) was first publicly released as an open-source network forensic analysis framework on GitHub in 2014, written in Python 2. In 2020 Dshell was rewritten in Python 3 from the ground up and again made available as open-source software on GitHub, following the Python 2 language deprecation on 1 JAN 2020.\(^2\) Plugins written for the deprecated Python 2 version of Dshell are not compatible with this version and vice versa. The Dshell\(^1\) GitHub repository contains the current Python 3 version as well as an archived Python 2 version available as a tarball. This developer guide only applies to the current version.

Dshell is a tool for network forensic analysis that can be used out of the box for simple and advanced analyses, or customized to fit an end-user’s needs. Custom Dshell plugins can be developed to parse and analyze unique network traffic protocols and data, such as malware. Existing plugins can be modified to extract different information from the protocols they currently parse, customize the programmatic actions performed on the data, or alter the outputted information when using the plugin. For a detailed guide on using the Dshell framework—including getting started with it, using its full capabilities, and gaining a better understanding of the framework from a user’s perspective—please see the Dshell User Guide.\(^3\)

2. Important Concepts

The Dshell framework involves several concepts that should be understood for plugin development.

All information in this section is for background purposes to help developers better understand the inner workings of the Dshell framework. This section covers core
Dshell framework functionality, which is not intended to be altered by plugin developers.

2.1 Data Sources

Dshell can read packets from two types of sources: 1) pcap and pcapng files and 2) network interfaces. This report uses the term “data source” to reference either of these sources of packets in general terms.

The code for handling data sources is in dshell/decode.py within its main function. Here, the command line arguments passed when calling decode are parsed, including an argument setting the data source(s) as an interface or pcap file(s). The name of the data source can be accessed inside of a plugin from the PacketPlugin superclass’s current_pcap_file attribute.

Dshell uses the third-party library pcapy-ng to read packets from data sources. Each raw packet is processed by a third-party library pypacker, then further refined as a Dshell Packet object.

2.2 Plugin Chain and Produce/Consume Model

When users run Dshell, they can build a chain of sequential plugins to control and filter packets. In practice, users generally only use one plugin, meaning the chain will only contain that plugin.

Plugin developers define how those plugins in the chain decide which data are passed on to the next plugin in the chain. Handler functions in plugins must use return statements indicating whether a packet, connection, or similar will continue to the next plugin. The type of object(s) to return depends on the type of handler but will generally match the types of the handler’s input. Dshell will display a warning if the return values are not the right type.

The chain is handled in dshell/decode.py. It creates the plugin_chain list from user-provided arguments and uses its feed_plugin_chain function to send data source packets to the first plugin in that chain.

Inside the feed_plugin_chain function, each packet is fed to a plugin’s consume_packet function, defined in the PacketPlugin class in dshell/core.py. The packet is processed by the plugin, and any handler output is stored in an internal _packet_queue. The feed_plugin_chain function then takes the packets from that queue and recursively calls itself with the next plugin in the chain and each produced packet individually.
2.3 Parallelization

Users can choose to run Dshell in multiple processes using the -P or --parallel arguments. When run this way, Dshell divides the handling of each provided data source into separate Python processes. This is something to keep in mind when developing plugins that may handle overall state or need output in a strictly ordered format.

3. Core Object Classes

Dshell defines three classes that are used when handling data within plugins: Packet, Connection, and Blob. They are all defined in dshell/core.py alongside the two main plugin classes described later in this guide: PacketPlugin and ConnectionPlugin.

All three classes define an info function that generates an overview dictionary of information about an instantiation. This is most commonly used to populate arguments when calling a plugin’s write function, providing the information written out by the plugin to the user via the CLI.

3.1 Packet

After a packet is pulled from a data source and parsed by pypacker, it is further refined into a Packet object. When initialized, it attempts to populate several attributes based on the protocols used. The full list of attributes is available in the class’s docstring and Table 1. Additional information is also provided in bold on those that are less straightforward. Please refer to the List of Symbols, Abbreviations, and Acronyms at the end of this report for definitions of terminology used within the tables.
Table 1  Dshell packet object class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>Packet attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>dt</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>frame</td>
</tr>
<tr>
<td>pkt</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>rawpkt</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>pktlen</td>
</tr>
<tr>
<td>byte_count</td>
</tr>
<tr>
<td>sip</td>
</tr>
<tr>
<td>dip</td>
</tr>
<tr>
<td>sip_bytes</td>
</tr>
<tr>
<td>dip_bytes</td>
</tr>
<tr>
<td>sport</td>
</tr>
<tr>
<td>dport</td>
</tr>
<tr>
<td>smac</td>
</tr>
<tr>
<td>dmac</td>
</tr>
<tr>
<td>sipcc</td>
</tr>
<tr>
<td>dipcc</td>
</tr>
<tr>
<td>siplat</td>
</tr>
<tr>
<td>diplat</td>
</tr>
<tr>
<td>siplon</td>
</tr>
<tr>
<td>diplon</td>
</tr>
<tr>
<td>sipasn</td>
</tr>
<tr>
<td>dipasn</td>
</tr>
<tr>
<td>protocol</td>
</tr>
<tr>
<td>protocol_num</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>sequence_number</td>
</tr>
<tr>
<td>ack_number</td>
</tr>
<tr>
<td>tcp_flags</td>
</tr>
<tr>
<td>addr</td>
</tr>
<tr>
<td>byte_count</td>
</tr>
<tr>
<td>packet_tuple</td>
</tr>
<tr>
<td>rawpkt</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>info</td>
</tr>
</tbody>
</table>

---

[a] Attributes available via Python class @property decorators
[b] Class function that provides information about the class data
3.2 Connection

Connection objects are used to hold metadata about individual network connections, collect the network’s connection packets, and reassemble the data passed by those streams of packets. A connection is instantiated when the first packet of a new connection is handled. When initialized, it attempts to populate several attributes based on the protocols used. The full list of attributes is available in the class’s docstring and Table 2. Additional information about Dshell’s attribute terminology are also defined and discussed.

Table 2 Dshell connection object class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>Connection attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
</tr>
<tr>
<td>sip</td>
</tr>
<tr>
<td>smac</td>
</tr>
<tr>
<td>sport</td>
</tr>
<tr>
<td>sipcc</td>
</tr>
<tr>
<td>siplat</td>
</tr>
<tr>
<td>siplon</td>
</tr>
<tr>
<td>sipasn</td>
</tr>
<tr>
<td>clientip</td>
</tr>
<tr>
<td>clientmac</td>
</tr>
<tr>
<td>clientport</td>
</tr>
<tr>
<td>clientcc</td>
</tr>
<tr>
<td>clientlat</td>
</tr>
<tr>
<td>clientlon</td>
</tr>
<tr>
<td>clientasn</td>
</tr>
<tr>
<td>dip</td>
</tr>
<tr>
<td>dmac</td>
</tr>
<tr>
<td>dport</td>
</tr>
<tr>
<td>dipcc</td>
</tr>
<tr>
<td>diplat</td>
</tr>
<tr>
<td>diplon</td>
</tr>
<tr>
<td>dipasn</td>
</tr>
<tr>
<td>serverip</td>
</tr>
<tr>
<td>servermac</td>
</tr>
<tr>
<td>serverport</td>
</tr>
<tr>
<td>servercc</td>
</tr>
<tr>
<td>serverlat</td>
</tr>
<tr>
<td>serverlon</td>
</tr>
<tr>
<td>serverasn</td>
</tr>
<tr>
<td>protocol</td>
</tr>
<tr>
<td>clientpackets^</td>
</tr>
<tr>
<td>clientbytes^</td>
</tr>
<tr>
<td>serverpackets^</td>
</tr>
<tr>
<td>serverbytes^</td>
</tr>
<tr>
<td>ts</td>
</tr>
<tr>
<td>dt</td>
</tr>
<tr>
<td>starttime</td>
</tr>
<tr>
<td>Connection attributes</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td><strong>endtime</strong></td>
</tr>
<tr>
<td><strong>client_state</strong></td>
</tr>
<tr>
<td><strong>server_state</strong></td>
</tr>
<tr>
<td><strong>blobs</strong></td>
</tr>
<tr>
<td><strong>stop</strong></td>
</tr>
<tr>
<td><strong>handled</strong></td>
</tr>
<tr>
<td><strong>duration</strong></td>
</tr>
<tr>
<td><strong>closed (bool)</strong></td>
</tr>
<tr>
<td><strong>established (bool)</strong></td>
</tr>
<tr>
<td><strong>blobs</strong></td>
</tr>
<tr>
<td><strong>info</strong></td>
</tr>
</tbody>
</table>

\[a\] Attributes available via Python class @property decorators

\[b\] Class function that provides information about the class data

### 3.2.1 Server/Client Versus Destination/Source

For the functional purposes of Dshell, the source attributes and the concept of client attributes are interchangeable; the destination attributes and the concept of server attributes are also interchangeable.

### 3.2.2 Blobs

The Blob object is defined in Section 3.3, but the **blobs** attribute of a connection object is a dynamically populated iterator of reassembled groups of unidirectional packets from the connection. In general practice, the preferred method to deal with Blobs is by using the **ConnectionPlugin**’s **blob_handler** function.

### 3.3 Blob

When a connection streams one or more packets in a single direction (excluding TCP ACK and TCP handshake [SYN, SYN-ACK, ACK] packets), Dshell groups these packets into objects called Blobs for data reassembly. A new Blob is instantiated when a connection sees a data-containing packet moving in the opposite direction of the previous data-containing packet. This new directional packet is used to instantiate the new Blob. The full list of attributes is available in the class’s docstring and Table 3.
Table 3  Dshell Blob object class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>Blob attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>.addr attribute of the first packet</td>
</tr>
<tr>
<td>ts</td>
<td>timestamp of the first packet</td>
</tr>
<tr>
<td>starttime(^a)</td>
<td>datetime of first packet</td>
</tr>
<tr>
<td>endtime(^a)</td>
<td>datetime of last packet</td>
</tr>
<tr>
<td>sip</td>
<td>source IP</td>
</tr>
<tr>
<td>smac</td>
<td>source MAC address</td>
</tr>
<tr>
<td>sport</td>
<td>source port</td>
</tr>
<tr>
<td>sipasc</td>
<td>country code of source IP</td>
</tr>
<tr>
<td>sipasn</td>
<td>ASN of source IP</td>
</tr>
<tr>
<td>dip</td>
<td>dest IP</td>
</tr>
<tr>
<td>dmac</td>
<td>dest MAC address</td>
</tr>
<tr>
<td>dport</td>
<td>dest port</td>
</tr>
<tr>
<td>dipcc</td>
<td>country code of dest IP</td>
</tr>
<tr>
<td>dipasn</td>
<td>ASN of dest IP</td>
</tr>
<tr>
<td>protocol</td>
<td>text version of protocol in layer-3 header</td>
</tr>
<tr>
<td>direction</td>
<td>direction of the Blob - 'cs' for client-to-server, 'sc' for server-to-client</td>
</tr>
<tr>
<td>ack_sequence_numbers</td>
<td>set of ACK numbers from the receiver for collected data packets</td>
</tr>
<tr>
<td>packets</td>
<td>list of all packets in the Blob</td>
</tr>
<tr>
<td>hidden (bool)</td>
<td>Used to indicate that a Blob should not be passed to next plugin. Can theoretically be overruled in a connection_handler to force a Blob to be passed to next plugin.</td>
</tr>
<tr>
<td>all_packets(^a)</td>
<td>(deprecated, replaced with “packets” attribute) list of all packets in the Blob</td>
</tr>
<tr>
<td>start_time(^a)</td>
<td>(returns “starttime”) datetime for first packet</td>
</tr>
<tr>
<td>end_time(^a)</td>
<td>(returns “endtime”) datetime of last packet</td>
</tr>
<tr>
<td>frames(^a)</td>
<td>The frame identifiers for the packets that contain the message.</td>
</tr>
<tr>
<td>sequence_numbers(^a)</td>
<td>The starting sequence numbers found within the packets.</td>
</tr>
<tr>
<td>sequence_range(^a)</td>
<td>The range of sequence numbers found within the packets.</td>
</tr>
<tr>
<td>segments(^a)</td>
<td>List of valid (sequence number, packet) tuples in order by sequence number.</td>
</tr>
<tr>
<td>data(^a)</td>
<td>Raw data of TCP message.</td>
</tr>
<tr>
<td>reassemble(^a)</td>
<td>Rebuild the data string from the current list of data packets. For each packet, the TCP sequence number is checked. If overlapping or padding is disallowed, it will raise a SequenceNumberError exception if a respective event occurs. Allows additional options via the following arguments:</td>
</tr>
<tr>
<td>reassemble</td>
<td>allow_padding (bool): If data is missing and allow_padding = True (default: True), then the padding argument will be used to fill the gaps</td>
</tr>
<tr>
<td></td>
<td>allow_overlap (bool): If data is overlapping, the new data is used if the allow_overlap argument is True (default), otherwise, the earliest data is kept</td>
</tr>
<tr>
<td></td>
<td>padding: Byte character(s) to use to fill in missing data. Used in conjunction with allow_padding (default: b'\x00')</td>
</tr>
<tr>
<td>info(^b)</td>
<td>Provides a dictionary with information about a Blob. Useful for calls to a plugin’s write() function, e.g., self.write(*blob.info()).</td>
</tr>
</tbody>
</table>

\(^a\) Attributes available via Python class @property decorators
\(^b\) Class function that returns data
A Connection dynamically creates and handles its Blobs after it closes. No Blobs are cached. This is by design to allow out-of-order or retransmitted packets to be grouped into their appropriate Blobs.

The reassembled bytestring of a Blob’s data can be accessed via the data attribute for the raw data or by calling the Blob’s reassemble function, which provides additional options. By default, reassemble pads any missing data sections with null characters and overlaps any early data with later data. Instead of null characters, the padding character can be defined by providing a bytestring to the padding argument and setting the allow_padding argument to True. If the argument allow_padding is set to False, any missing data raises a dshell.core.SequenceNumberError exception. If the argument allow_overlap is set to False, any data that overlaps existing data raises a dshell.core.SequenceNumberError exception.

Note that the code defining Blob has several TO DO comments for updates to better handle edge cases involving partial, corrupted, or misleading connection data.

4. Example Plugin – netflow.py

The simplest plugin available is the Netflow plugin (dshell/plugins/flows/netflow.py). It simply tracks connections and collects basic information about them. That information is presented to the user as connections close.

```python
import dshell.core
from dshell.output.netflowout import NetflowOutput

class DshellPlugin(dshell.core.ConnectionPlugin):
    def __init__(self, *args, **kwargs):
        super().__init__(
            name="Netflow",
            description="Collects and displays statistics about connections",
            author="dev195",
            bpf="ip or ip6",
            output=NetflowOutput(label=\__name\__),
        )

    def connection_handler(self, self, conn):
        self.write(**conn.info())
        return conn
```

The two initial import statements: the core Dshell library (dshell.core) with its definitions for plugins classes and other objects, and an output module that is used to set the default output format.
The class definition follows the import statements. For plugin scripts, Dshell looks specifically for a class named “DshellPlugin.” It must inherit from one of the core plugin classes, `PacketPlugin` or `ConnectionPlugin`, or one their derivative subclasses, such as `HTTPPlugin`.

The initialization function, `__init__`, is flexible, but should start by calling the superclass’s `__init__` function with values for both the required and optional developer-defined fields if applicable. Table 4 shows these required and common (but optional) attributes that the Netflow plugin defines, as well as two it does not.

Table 4  Dshell PacketPlugin public class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>Required and common (but optional) attributes used by plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>name(^a)</td>
</tr>
<tr>
<td>description(^a)</td>
</tr>
<tr>
<td>author(^a)</td>
</tr>
<tr>
<td>bpf(^b)</td>
</tr>
<tr>
<td>output(^b)</td>
</tr>
<tr>
<td>longdescription(^b)</td>
</tr>
<tr>
<td>optiondict(^b)</td>
</tr>
</tbody>
</table>

\(^a\) Developer-defined values

\(^b\) Optional developer-defined values

Finally, the plugin defines its key handler function, `connection_handler`. Handler functions are described in the following sections, but the Netflow plugin uses it to simply output the information Dshell has collected about a completed connection.

5. Plugin Types

Dshell includes two plugin superclasses and several generic subclasses to inherit when developing new plugins.

5.1 PacketPlugin

`PacketPlugin` is the base plugin class that all others inherit from and is defined in `dshell.core`. This plugin is used to handle individual packets. To handle reconstructed connections, use the `ConnectionPlugin` instead.
The full list of attributes is available in the class’s docstring as well as in Table 5. Attributes that are not developer-defined are automatically populated as the plugin runs and should be treated as read only.

Table 5  Dshell PacketPlugin public class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>PacketPlugin public attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name⁶</td>
</tr>
<tr>
<td>description⁶</td>
</tr>
<tr>
<td>longdescription⁷</td>
</tr>
<tr>
<td>bpf⁸</td>
</tr>
<tr>
<td>compiled_bpf</td>
</tr>
<tr>
<td>vlan_bpf</td>
</tr>
<tr>
<td>author⁹</td>
</tr>
<tr>
<td>seen_packet_count</td>
</tr>
<tr>
<td>handled_packet_count</td>
</tr>
<tr>
<td>seen_conn_count</td>
</tr>
<tr>
<td>handled_conn_count</td>
</tr>
<tr>
<td>optiondict⁸</td>
</tr>
<tr>
<td>out⁹</td>
</tr>
<tr>
<td>link_layer_type</td>
</tr>
<tr>
<td>defrag_ip</td>
</tr>
<tr>
<td>logger</td>
</tr>
<tr>
<td>current_pcap_file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>the name of the plugin as presented to the user in help text and logs</td>
</tr>
<tr>
<td>short description of the plugin (used with decode -l)</td>
</tr>
<tr>
<td>verbose description of the plugin (used with -h). Defaults to description</td>
</tr>
<tr>
<td>value, if not provided.</td>
</tr>
<tr>
<td>the initial Berkeley Packet Filter (BPF) to apply to traffic entering</td>
</tr>
<tr>
<td>plugin</td>
</tr>
<tr>
<td>a compiled BPF for pcap, usually created in decode.py</td>
</tr>
<tr>
<td>Boolean that tells whether BPF should be compiled with VLAN support</td>
</tr>
<tr>
<td>the plugin author, traditionally written using initials</td>
</tr>
<tr>
<td>number of packets this plugin has seen</td>
</tr>
<tr>
<td>number of packets this plugin has passed through a handler function</td>
</tr>
<tr>
<td>number of connections this plugin has seen</td>
</tr>
<tr>
<td>number of connections this plugin has passed through a handler function</td>
</tr>
<tr>
<td>dict of options specific to this plugin, provided in the following format</td>
</tr>
<tr>
<td>'optname'{configdict} translates to –pluginname_optname</td>
</tr>
<tr>
<td>output module instance, provided using “output” argument</td>
</tr>
<tr>
<td>numeric label for link layer of current data source</td>
</tr>
<tr>
<td>rebuild fragmented IP packets (default True)</td>
</tr>
<tr>
<td>plugin-specific logging object for printing log messages</td>
</tr>
<tr>
<td>string containing the name of the data source, either an interface name or</td>
</tr>
<tr>
<td>pcap file path</td>
</tr>
</tbody>
</table>

⁶ Developer-defined values  
⁷ Optional developer-defined values

5.1.1 Placeholder Functions

Most functions for the class are meant only for internal use within the class, but several placeholder functions are defined that can be overwritten by subclasses (custom Dshell plugins) depending on their needs.

- **packet_handler**: A function called for every packet from a data source. It receives one argument, `pkt`, a Packet object. Plugins use this function to interact with packets. To pass packets back into the plugin chain, it should return a Packet object.

- **premodule**: A function called before any data is pulled into the framework. It is generally used for initialization tasks, such as setting up state, reading data files, making application programming interface (API) connections, and so forth.
- **prefile**: A function called before a file or interface is processed, but after `premodule` is called. It receives one argument, `infile`, the string filepath or interface of the data source. If using multiple data files, this function is called before each. It is generally used for initialization tasks, such as zeroing counters, printing debug information, and so forth.

- **postfile**: A function called immediately after a data file is closed. If using multiple data files, this function is called after each. It is generally used for cleanup tasks, such as printing file statistics, closing file handles, and so forth.

- **postmodule**: A function called after all files and data are processed and the framework is preparing to close. It is generally used for final cleanup tasks, such as printing overall statistics, printing debug information, closing API connections, and so forth.

- **filter**: A function to determine if a packet should be accepted or dropped by the plugin. By default, uses the plugin’s defined BPF. It receives one argument, the Packet object, and must return either True (accept the packet) or False (drop the packet). It is run after a packet passes through the BPF. It is generally used for defining more complex filters that cannot be defined in a BPF, such as conditional logic. Defining a complex filter this way may significantly increase latency when processing live packet capture.

### 5.1.2 Other Functions

`PacketPlugin` also exposes some additional functions for use by plugin developers inside their plugins but should not be overwritten.

- **write**: A function used to send output to an output module for user consumption. It has no defined set of expected arguments but should be passed as much information as is relevant for the plugin being developed. Most common arguments for several output modules can be provided by unpacking (using `**` when passing to the function) the `info` function of Packet, Blob, or Connection objects.

- **recompile_bpf**: A function used to recompile the `bpf` attribute as a pcap-compatible object. It should be called when a plugin updates its BPF string during runtime, such as the ftp plugin dynamically changing its BPF to allow the processing of data transfer channels established during the connection.
5.2 ConnectionPlugin

ConnectionPlugin is a subclass of PacketPlugin and is intended for handling reconstructed transmission control protocol (TCP) and user datagram protocol (UDP) connections. It contains the same attributes as PacketPlugin, but adds a few additional public attributes for its expanded purpose. The full list of attributes is available in the class’s docstring and Table 6.

Table 6 Dshell ConnectionPlugin public class attributes with descriptions, defined in dshell/core.py

<table>
<thead>
<tr>
<th>ConnectionPlugin public attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seen_conn_count</td>
<td>how many new connections were seen by the plugin</td>
</tr>
<tr>
<td>handled_conn_count</td>
<td>how many connections were fully handled by the plugin</td>
</tr>
<tr>
<td>mixbloods</td>
<td>maximum number of blobs a connection will store before calling connection_handler</td>
</tr>
<tr>
<td>timeout</td>
<td>how long do we wait before deciding a connection is “finished.” Time is checked by iterating over cached connections and checking if the timestamp of the connection’s last packet is older than the timestamp of the current packet, minus this value.</td>
</tr>
<tr>
<td>timeout_frequency</td>
<td>The number of packets to process between timeout checks</td>
</tr>
<tr>
<td>max_open_connections</td>
<td>The maximum number of open connections allowed at one time. If the maximum number of connections is met, the oldest connections will be force closed.</td>
</tr>
</tbody>
</table>

It also updates its inherited produce_packets function by waiting for connections to be successfully handled before yielding any packets inside those connections.

Placeholder Functions

Most functions for the class are meant only for internal use within the function, but several additional placeholder functions are defined beyond those of the PacketPlugin superclass that can be overwritten by subclasses (custom Dshell plugins) depending on their needs.

- connection_handler: A function called once a connection is considered closed, passes a time boundary, or passes a defined packet count threshold. It receives one argument, conn, a Connection object. It is generally used to interact with reassembled connections. To pass packets back into the plugin chain, it should return a Connection object. Any Blobs in the Connection with their hidden attribute set to True will not have their packets continue through the plugin chain, even if they are part of a Connection that is returned. This could be used, for example, to filter out one side of the connection.
• **connection_init_handler**: A function called when the first packet of a new connection is seen, but after it passes through `packet_handler`. It receives one argument, `conn`, the newly created `Connection` object. It returns nothing. It is generally used for initialization tasks, such as zeroing counters, printing debug information, first pass filtering, and so forth.

• **connection_close_handler**: A function called when a TCP connection is properly closed with RST or FIN packets. It receives one argument, `conn`, the newly created `Connection` object. It returns nothing. It is generally used for cleanup tasks, such as printing or storing connection statistics. Because this function is called only when a connection properly closes, it may miss connections that time out, get cut off, or do not end before a data source finishes processing. If connections must be handled, use the `connection_handler` function. The function, `connection_handler`, is called on all hanging connections when a data source is closed.

• **blob_handler**: A function called when a connection closes; its packets are chunked into groups based on stream direction. It receives two arguments—`conn` and `blob`, a `Connection` object, and a `Blob` object, respectively. It should return two values, a `Connection` and a `Blob`, or nothing when a Blob’s packets should not continue along the plugin chain. It is generally used when a plugin is only interested in reassembled parts of a connection, such as the stream of data from a server to client or vice versa.

### 5.3 DNSPlugin

A **DNSPlugin** is a subclass of the **ConnectionPlugin**. It is defined in `dshell.plugins.dnsplugin`. It is meant to ease the handling of Domain Name System (DNS) requests and responses. Any packets that are not associated with DNS are not handled or passed back into the plugin chain.

**Placeholder Functions**

Alongside all the functions inherited from its parent classes, it defines an additional placeholder function: `dns_handler`.

**DNSPlugin** defines its own internal `connection_handler` function; therefore, plugins should not overwrite it when inheriting. The handler sorts DNS packets in a connection into requests and responses, and pairs them by their ID numbers. The request-and-response pairs are passed to `dns_handler` by each ID group.

• **dns_handler**: A function called when a connection is handled (see `connection_handler` in Section 5.2). It receives three arguments:
1) conn: the Connection containing the DNS traffic
2) requests: a list of packets flagged as DNS requests, or None
3) responses: a list of packets flagged as DNS responses associated with request packets, or None

If the packets continue along the plugin chain, this function should return what it received as arguments: conn, requests, and responses.

5.4 HTTPPlugin

An HTTPPlugin is a subclass of the ConnectionPlugin. It is defined in dshell.plugins.httpplugin. It is meant to ease the handling of hypertext transfer protocol (HTTP) requests and responses. Any packets that are not associated with an HTTP request or response are not handled or passed back into the plugin chain.

5.4.1 Placeholder Functions

Alongside all the functions inherited from its parent classes, it defines an additional placeholder function: http_handler.

HTTPPlugin defines its own internal connection_handler function; therefore, plugins should not overwrite it when inheriting. It separates Blobs by their direction and attempts to convert them into HTTPRequest (for client-to-server Blobs) or HTTPResponse (for server-to-client Blobs) objects, explained as follows.

- http_handler: A function called when a connection is handled (see connection_handler in Section 5.2). It receives three arguments: conn (Connection), request (HTTPRequest), and response (HTTPResponse).

If the packets continue along the plugin chain, this function should return what it received as arguments: conn, request, and response.

5.4.2 Custom Classes: HTTPRequest and HTTPResponse

HTTPPlugin defines two new classes for internal use: HTTPRequest and HTTPResponse. The two are very similar in how they are constructed but provide different attributes. The full list of attributes is available in each class’s docstring as well as Tables 7 and 8.
Table 7 Dshell HTTPRequest class attributes with descriptions, defined in dshell/plugins/httpplugin.py

<table>
<thead>
<tr>
<th>HTTPRequest attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>the Blob instance of the request</td>
</tr>
<tr>
<td>errors</td>
<td>a list of caught exceptions from parsing</td>
</tr>
<tr>
<td>method</td>
<td>the method of the request (GET, PUT, POST, etc.)</td>
</tr>
<tr>
<td>uri</td>
<td>the URI being requested (host not included)</td>
</tr>
<tr>
<td>version</td>
<td>the HTTP version (e.g., “1.1” for “HTTP/1.1”)</td>
</tr>
<tr>
<td>headers</td>
<td>a dictionary containing the headers and values</td>
</tr>
<tr>
<td>body</td>
<td>bytestring of the reassembled body, after the headers</td>
</tr>
</tbody>
</table>

Table 8 Dshell HTTPResponse class attributes with descriptions, defined in dshell/plugins/httpplugin.py

<table>
<thead>
<tr>
<th>HTTPResponse attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>the Blob instance of the request</td>
</tr>
<tr>
<td>errors</td>
<td>a list of caught exceptions from parsing</td>
</tr>
<tr>
<td>version</td>
<td>the HTTP version (e.g., “1.1” for “HTTP/1.1”)</td>
</tr>
<tr>
<td>status</td>
<td>the status code of the response (e.g., “200” or “304”)</td>
</tr>
<tr>
<td>reason</td>
<td>the status text of the response (e.g., “OK” or “Not Modified”)</td>
</tr>
<tr>
<td>headers</td>
<td>a dictionary containing the headers and values</td>
</tr>
<tr>
<td>body</td>
<td>bytestring of the reassembled body, after the headers</td>
</tr>
</tbody>
</table>

Headers are parsed as key-value pairs and stored as a dictionary.

Both classes attempt to parse the body content from the stream. If content length is available, it attempts to reconstruct the data, and creates a dshell.core.DataError exception if data is missing. Any DataError exceptions are pushed to the class’s errors attribute. Handler functions can decide to handle or raise any of the exceptions stored in the errors attribute.

6. Building a Plugin

There is not a rigidly defined, step-by-step process for creating a new Dshell plugin. However, there are required parts of a plugin script that must be defined and developed. These parts can form a pipeline for developing a plugin, as shown in Table 9.
During development, plugins can be stored and tested short-term in the “plugins” directory of Dshell’s installation location, usually located in Python’s “site-packages” directory. This allows Dshell to find and use them directly. Plugins placed in this location may be overwritten when updating Dshell; therefore, it is encouraged to also back them up outside of this directory. Alternatively, plugins can be developed in a locally downloaded copy of Dshell. Calling Dshell with Python directly from within the local directory should find the local plugins, `python3 -m dshell.decode`. Lastly, a Dshell plugin pack can be created to store custom Dshell plugins. See Section 8 for instructions on setting up and using this method.

<table>
<thead>
<tr>
<th>Table 9</th>
<th>Overview of steps to build a Dshell plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decide purpose and metadata</strong></td>
<td>Most importantly, define the purpose of the plugin. Then, decide on the values for the developer-defined fields that make up the plugin, such as a unique name, an author, a description, and other fields defined in Table 5. Additionally, a default output module should be chosen if the plugin provides output to a user.</td>
</tr>
<tr>
<td><strong>Choose a parent plugin</strong></td>
<td>Choose one of the plugin superclasses defined in dshell.core, ConnectionPlugin or PacketPlugin, or one of their derivatives, such as DNSPlugin or HTTPPlugin. This sets most of the attributes and functions needed for Dshell to find and use the new plugin. It also provides a basic idea of how the plugin will handle the data source, either packet-by-packet in a PacketPlugin or connection-by-connection in the other plugin types.</td>
</tr>
<tr>
<td><strong>Define <strong>init</strong></strong></td>
<td>Define how the plugin will initialize itself in its <strong>init</strong> function. This should include a call to the superclass’s <strong>init</strong> function with values providing the metadata from the previous step.</td>
</tr>
<tr>
<td><strong>Define handlers</strong></td>
<td>A plugin will do most of its work in handler functions, such as packet_handler and connection_handler. These functions are also where calls to write will likely be placed, if applicable.</td>
</tr>
</tbody>
</table>

### 6.1 Building an Example Plugin

This section provides an example of building a toy plugin using the steps defined in Table 9.

#### 6.1.1 Decide Purpose and Metadata

The goal of this toy plugin is to find and count the number of instances of “.com” in data ingested from a data source. It will separate these counts between connections, data sources, and a final grand total.

Since this plugin is an example, we will name it “Example” and define additional metadata in Table 10.
Table 10  “Example” plugin metadata definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>test</td>
</tr>
<tr>
<td>BPF</td>
<td>tcp or udp</td>
</tr>
<tr>
<td>Description</td>
<td>A plugin to find and count the number of instances of “.com” in connections</td>
</tr>
<tr>
<td>Long description</td>
<td>A plugin development example. This plugin finds and counts the number of instances of “.com” in connections using most of the handler functions available to a ConnectionPlugin subclass.</td>
</tr>
</tbody>
</table>

The output from this plugin is simple, so we will use the default Output module.

Finally, we will want a user flag to decide if connections without any instances of “.com” should appear.

6.1.2 Pick a Parent Plugin

Since this plugin will be grouping results by connection, it will inherit from the ConnectionPlugin superclass instead of PacketPlugin. This will require importing the dshell.core module to access the class.

```python
import dshell.core
from dshell.output.output import Output
class DshellPlugin(dshell.core.ConnectionPlugin):
```

6.1.3 Define __init__

The initialization function of the plugin will use the metadata defined in the first step and pass it to the superclass’s __init__ function. Included in the call is the definition of the optiondict. Not all plugins will define the argument, but this example plugin will allow users to set a flag if they want to see information on connections and files that did not have instances of “.com.” The key is “show_zeroes,” which Dshell will automatically convert into the command-line argument --example_show_zeroes.

After setting the metadata, the plugin will initialize the attributes that store counts.

```python
def __init__(self):
    super().__init__(
        name="Example",
        author="test",
        bpf="tcp or udp",
        description="A plugin to find and count the number of instances of ".com" in connections",
        longdescription="A plugin development example. This plugin finds and counts the number of instances of ".com" in connections using
```
most of the handler functions available to a ConnectionPlugin subclass.

```python
output=Output(label=\_name\_),
optiondict={
    "show_zeroes": {
        "action": "store_true",
        "help": "Show connections without \".com\"",
        "default": False
    }
}
}
self.total_com_instances = 0
self.file_com_instances = {}
self.conn_com_instances = {}
```

### 6.1.4 Define Handlers

The functionality of the plugin is defined in its handler functions. In this example, we will use almost all of them for demonstration purposes. Most of the handler functions in this example plugin simply initialize counters or print results.

The `premodule` and `postmodule` functions set and write the total number of "com" instances.

```python
def premodule(self):
    self.total_com_instances = 0
def postmodule(self):
    self.write(".com seen {} total times".format(self.total_com_instances))
```

The `prefile` and `postfile` functions set and write the total number of "com" instances in individual files as they are opened and closed, respectively. The `postfile` additionally checks the `show_zeroes` flag to determine if it should print empty counters.

```python
def prefile(self, infile):
    self.file_com_instances[infile] = 0
def postfile(self):
    coms = self.file_com_instances.get(self.current_pcap_file, 0)
    if coms or self.show_zeroes:
        self.write(".com seen {} times in {}".format(coms, self.current_pcap_file))
```

The `connection_init_handler` creates a record for each connection as it begins and sets it to 0.

```python
def connection_init_handler(self, conn):
    self.conn_com_instances[conn] = 0
```
The `connection_handler` function, called near the end of connections, prints count messages for each connection if any “.com” was seen or if the `show_zeros` flag is set to True. Additionally, it provides the `conn.info()` output to the `write` function for output modules that can use it. It also acts as a final filter, returning nothing if no “.com” was found and preventing such connections from continuing along the plugin chain.

```python
def connection_handler(self, conn):
    coms = self.conn_com_instances.get(conn, 0)
    if coms or self.show_zeros:
        msg = f".com seen {coms} times in ({conn.sip}:{conn.sport} -> {conn.dip}:{conn.dport})"
        self.write(msg, **conn.info())
    return conn
else:
    return
```

The `blob_handler` is the function that counts actual “.com” instances. It reassembles the stream of data with `blob.data` and uses Python’s built-in `count` function to tally each “.com” in the string of bytes and adds the total to each counter. If “.com” is not found, it returns nothing, indicating to Dshell that the blob should not be passed along to the next plugin in the plugin chain.

```python
def blob_handler(self, conn, blob):
    coms = blob.data.lower().count(b'.com')
    if coms:
        self.conn_com_instances[conn] += coms
        self.file_com_instances[self.current_pcap_file] += coms
        self.total_com_instances += coms
        return conn, blob
    else:
        return
```

Put together, the example.py plugin will look like this:

```python
import dshell.core
from dshell.output.output import Output
class DshellPlugin(dshell.core.ConnectionPlugin):
    def __init__(self):
        super().__init__(
            name="Example",
            author="test",
            bpf="tcp or udp",
            description="A plugin to find and count the number of instances of ".com" in connections",
            longdescription="A plugin development example. This plugin finds and counts the number of instances of ".com" in connections using most of the handler functions available to a ConnectionPlugin subclass.",
            output=Output(label=__name__),
```
optiondict=
    "show_zeroes": {
        "action": "store_true",
        "help": "Show connections without ".com\"",
        "default": False
    }
}

self.total_com_instances = 0
self.file_com_instances = {}
self.conn_com_instances = {}

def premodule(self):
    self.total_com_instances = 0

def postmodule(self):
    self.write(".com seen {} total times".format(self.total_com_instances))

def prefile(self, infile):
    self.file_com_instances[infile] = 0

def postfile(self):
    coms = self.file_com_instances.get(self.current_pcap_file, 0)
    if coms or self.show_zeroes:
        self.write(".com seen {} times in {}".format(coms, self.current_pcap_file))

def connection_init_handler(self, conn):
    self.conn_com_instances[conn] = 0

def connection_handler(self, conn):
    coms = self.conn_com_instances.get(conn, 0)
    if coms or self.show_zeroes:
        msg = ".com seen {} times in ({}:{}) - ({}:{})".format(coms, conn.sip, conn.sport, conn.dip, conn.dport)
        self.write(msg, **conn.info())
        return conn
    else:
        return

def blob_handler(self, conn, blob):
    coms = blob.data.lower().count(b'.com')
    if coms:
        self.conn_com_instances[conn] += coms
        self.file_com_instances[self.current_pcap_file] += coms
        self.total_com_instances += coms
        return conn, blob
    else:
        return

To test the plugin, the example.py script should be placed in one of the plugin directories where Dshell can find it, such as dshell/plugins/misc/. It can then
be called like any other Dshell plugin with `decode -p example`. Additionally, a
user can use the `--example_show_zeroes` flag to display output for connections
and files without “.com.”

### 6.1.5 Example Plugin Output Using Sample Traffic

```
Dshell> decode -p example ~/pcap/http_with_jpegs.cap
.com seen 5 times in (10.1.1.101:3179 -> 209.225.11.237:80)
.com seen 5 times in (10.1.1.101:3183 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3184 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3185 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3187 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3191 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3192 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3193 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3194 -> 209.225.0.6:80)
.com seen 29 times in /home/pcap/http_with_jpegs.cap
.com seen 29 total times
```

### 6.1.6 Example Pugin Output Using Custom show_zeroes Option and Sample Traffic

```
Dshell> decode -p example --example_show_zeroes
~/pcap/http_with_jpegs.cap
.com seen 0 times in (10.1.1.101:3177 -> 10.1.1.1:80)
.com seen 5 times in (10.1.1.101:3179 -> 209.225.11.237:80)
.com seen 5 times in (10.1.1.101:3183 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3184 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3185 -> 209.225.0.6:80)
.com seen 5 times in (10.1.1.101:3187 -> 209.225.0.6:80)
.com seen 0 times in (10.1.1.101:3188 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3189 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3190 -> 10.1.1.1:80)
.com seen 1 times in (10.1.1.101:3191 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3192 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3193 -> 209.225.0.6:80)
.com seen 1 times in (10.1.1.101:3194 -> 209.225.0.6:80)
.com seen 0 times in (10.1.1.101:3195 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3196 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3197 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3198 -> 10.1.1.1:80)
.com seen 0 times in (10.1.1.101:3199 -> 10.1.1.1:80)
.com seen 0 times in (209.225.11.237:None -> 10.1.1.101:None)
.com seen 0 times in (209.225.0.6:None -> 10.1.1.101:None)
.com seen 0 times in (209.225.0.6:31045 -> 10.1.1.101:26678)
.com seen 0 times in (209.225.0.6:19809 -> 10.1.1.101:12634)
.com seen 0 times in (209.225.0.6:19459 -> 10.1.1.101:15552)
.com seen 0 times in (209.225.0.6:26764 -> 10.1.1.101:8165)
.com seen 0 times in (209.225.0.6:10380 -> 10.1.1.101:49560)
.com seen 0 times in (209.225.0.6:26674 -> 10.1.1.101:25308)
.com seen 0 times in (10.1.1.101:3200 -> 10.1.1.1:80)
.com seen 29 times in /home/pcap/http_with_jpegs.cap
.com seen 29 total times
```
7. Other Example Plugins

7.1 Building a Plugin to Extract Key Data from a Known Protocol

This section provides an example of building a toy plugin using the steps defined in Table 9.

7.1.1 Decide Purpose and Metadata

The goal of this toy plugin will be to extract the “Referer” field from HTTP web traffic sessions. This is an optional field that stores the IP address of the webpage that the user was referred from, and in this way provides a history of the user’s web browsing.

7.1.2 Pick a Parent Plugin

Since this plugin will be extracting data from HTTP web traffic, the parent plugin HTTPPlugin can be used, which is a subclass of the ConnectionPlugin and attempts to parse HTTP Requests and Responses and store their data in dictionaries.

7.1.3 Define __init__

After providing the name, author, and description of the plugin, the BPF should be set. Following the precedent set by the HTTP plugins native to the Dshell framework, the BPF can be set to “tcp and (port 80 or port 8080 or port 8000)” to filter on ports most commonly used for web traffic. Next, looking over the output options, and those used by other HTTP plugins native to Dshell (web, httpdump, riphttp) the alertout output module will provide enough detailed information on the current Blob that the HTTP session is within, so that is chosen and defined.

7.1.4 Define Handlers

The HTTPPlugin parent class defines the http_handler function that can be used to access the connection data, as well as HTTP request and response data dictionaries. The “Referer” field is an HTTP header field unique to HTTP Requests; therefore, a conditional is defined first to only process a request. Next, since the http_handler has already attempted to parse this information from the HTTP Request, simply use the Python dictionary .get() method to pull this data if it exists or a default value otherwise. Finally a call to self.write() with the referrer data and the unpacked data stored in request.blob.info() will provide detailed output information on the current blob and the referrer data. Additional data can be extracted from the HTTP request to provide the end user with a big picture understanding of the web traffic, such as verbose output including HTTP
Request fields detailing websites reached and the referrer to those websites. Pulling the HTTP Request method, host, and Uniform Resource Identifier (URI) fields in addition to the “Referer” field will provide this big picture understanding.

Put together, the referer.py plugin will look like this:

```python
# referer.py
import dshell.core
from dshell.plugins.httpplugin import HTTPPlugin
from dshell.output.alertout import AlertOutput

class DshellPlugin(HTTPPlugin):
    def __init__(self):
        super().__init__(
            name="referer",
            author="dek",
            description="Extract Referer information from HTTP sessions",
            bpf="tcp and (port 80 or port 8080 or port 8000)",
            output=AlertOutput(label=__name__),
            optiondict={
                's': {
                    'action': "store_true",
                    'default': None,
                    'help': 'show simple output to just pull "referer" field, without providing verbose HTTP Request fields details'
                }
            }
        )

    def http_handler(self, conn, request, response):
        if request:
            referer = request.headers.get('referer', None)

            # Simple output to just pull 'referer' field
            if self.s:
                self.write(f'\treferer:{referer}', **request.blob.info())

            # Verbose output (default) including HTTP Request fields detailing
            # websites reached and the "referer" to those websites
            else:
                method = request.method
                host = request.headers.get('host', '')
                uri = request.uri
                self.write(f'\t{method} {host}{uri} [referer: {referer}]', **request.blob.info())
        return conn, request, response
```
7.1.5 Referer Plugin Output Using Sample Traffic (Truncated)

Dshell> decode -p referer /home/pcap/http_with_jpegs.cap

...

7.1.6 Referer Plugin Output Using Custom Simple Option and Sample Traffic (Truncated)

Dshell> decode -p referer --referer_s /home/pcap/http_with_jpegs.cap

...

7.2 Building a Plugin to Decode Data from a Custom Protocol

This section provides an example of building a toy plugin using the steps defined in Table 9.

7.2.1 Decide Purpose and Metadata

The goal of this toy plugin will be to extract command and control (C2) messages sent between malware on an infected victim machine and a command server, as done in the workshop and cybersecurity study, *Uncovering and Decoding Malware Communications with Dshell*. Through analysis of the traffic, analysts determined
the messages were obfuscated with a simple ROT13 rotational cipher, a type of Caesar cipher that shifts the letters by 13 places forward from their normal location in the 26 character English alphabet. The ROT13 cipher is a unique cipher in that by performing the obfuscation twice, the original text is obtained as it shifts the characters perfectly back to their original locations in the English alphabet.

7.2.2 Pick a Parent Plugin

Since this plugin will be extracting reassembled data from any connection and the key idea is to obtain a big picture understanding of the C2 messages, it is best to work at the connection and Blob levels, so a ConnectionPlugin will be used.

7.2.3 Define __init__

After providing the name, author, and description of the plugin, the BPF should be set. In order to keep the filtering very broad, the BPF can be set to “ip or ip6.” Next, looking over the output options, the netflowout output module will provide detailed information on the connection, so that is chosen and defined.

7.2.4 Define Handlers

In order to view all of the C2 messages in one direction of the connection, the data should be parsed at the Blob level, so the blob_handler can be used. Making use of Python’s built-in str.maketrans() function, a translation can be defined that replaces any character seen in the first argument string with the character in the same location of the second argument string. This translation will be a rotational translation of 13 characters to implement the ROT13 cipher obfuscation and decoding. The raw C2 messages are obtained from the Blob data and then the custom ROT13 translation is applied to obtain the decoded messages. Finally, the self.write() method is called to output the message along with the general information from the connection provided by unpacking the data stored in conn.info(). Put together, the rot13.py plugin will look like this:

```python
# rot13.py
""
Decodes malware C2 messages obfuscated by ROT13 cipher
""

import dshell.core
from dshell.output.netflowout import NetflowOutput

class DshellPlugin(dshell.core.ConnectionPlugin):
    def __init__(self, *args, **kwargs):
        super().__init__(
            name="ROT13 C2 Decoder",
            description="Decodes malware C2 messages obfuscated by
ROT13",
```
7.2.5 Rot13 Plugin Output Using Example Traffic

Dshell> decode -p rot13 /home/pcap/rot13_example.pcap

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Source IP</th>
<th>Destination IP</th>
<th>Flags</th>
<th>Bytes Sent</th>
<th>Bytes Received</th>
<th>Time (s)</th>
<th>Bites/Sec Sent</th>
<th>Bites/Sec Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>2023-02-27</td>
<td>3:14:15</td>
<td>10.0.0.3</td>
<td>10.0.0.4</td>
<td>TCP</td>
<td>12345</td>
<td>23.4567</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>obfuscated:</td>
<td>b'png /rgp/cnffjq'</td>
<td></td>
<td>decoded:</td>
<td>o'cat /etc/passwd'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2023-02-27</td>
<td>3:14:16</td>
<td>10.0.0.3</td>
<td>10.0.0.4</td>
<td>TCP</td>
<td>12345</td>
<td>23.4567</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>obfuscated:</td>
<td>b'pbzznaq rkrphgrq'</td>
<td></td>
<td>decoded:</td>
<td>o'command executed'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.3 Modifying an Existing Plugin

This section provides an example of building a toy plugin using the steps defined in Table 9.

7.3.1 Decide Purpose and Metadata

The goal of this toy plugin is to collect and display statistics about connections using custom connection timeout logic, different than that used in the netflow plugin. The existing netflow plugin can simply be copied and its existing connection logic can be modified.

7.3.2 Pick a Parent Plugin

Since only the existing netflow plugin’s connection logic will be modified, the parent plugin will be kept as a ConnectionPlugin.

7.3.3 Define __init__

After providing the name, author, and description of the plugin, the BPF and output module should be set. Since only the existing netflow plugin’s connection logic will be modified, the BPF and output module will remain unchanged. Following the super().__init__() call the connection logic inherited by the parent plugin...
and stored in `self` can be modified, including the `self.timeout`, `self.timeout_frequency`, and `self.max_open_connections`. For an example—such as to better analyze the traffic commonly seen by a sensor—these are modified as follows: `self.timeout` reduced from 1 h to 1 s, `self.timeout_frequency` decreased from default of processing 300 packets before checking for timeout to just 1 packet, and `self.max_open_connections` increased from 1,000 to 10,000.

### 7.3.4 Define Handlers

Since only the existing netflow plugin’s connection logic will be modified, and this has been accomplished in the `__init__` function, the handler will remain unchanged.

Put together, the netflow-ct.py plugin will look like this:

```python
# netflow_ct.py
import dshell.core
from dshell.output.netflowout import NetflowOutput

# Added to update Connection timeout logic
import datetime

class DshellPlugin(dshell.core.ConnectionPlugin):
    def __init__(self, *args, **kwargs):
        super().__init__(
            name="Netflow Custom Timeout",
            description="Collects and displays statistics about
            connections, using custom Connection timeout logic",
            author="dek",
            bpf="ip or ip6",
            output=NetflowOutput(label=__name__),
        )

        # Update Connection timeout logic to better handle custom needs
        #   Connection timeout, decreased from default of 1 hour
        self.timeout = datetime.timedelta(seconds=1)
        #   Packets to process before checking for timeout,
        #   decreased from default of 300
        self.timeout_frequency = 1
        #   Maximum number of connections allowed,
        #   increased from default of 1000
        self.max_open_connections = 10000

    def connection_handler(self, conn):
        self.write(**conn.info())
        return conn
```
7.3.5 Netflow_ct Plugin Output Using Sample Traffic (Truncated)

Dshell> decode -p netflow_ct ~/pcap/http_with_jpegs.cap
2004-11-19 17:29:14          10.1.1.101 ->         10.1.1.1  (-- -> --)
TCP    3177      80     1      1      476      435  0.1368s
2004-11-19 17:29:15          10.1.1.101 ->         10.1.1.1  (-- -> --)
TCP    3188      80     1      4      574     4601  0.1278s
TCP    3179      80     2      2      993     1224  1.3282s
2004-11-19 17:29:15          10.1.1.101 ->         10.1.1.1  (-- -> --)
TCP    3189      80     1      6      597     8566  0.1643s
2004-11-19 17:29:15          10.1.1.101 ->         10.1.1.1  (-- -> --)
TCP    3190      80     1      7      600     9330  0.3300s
TCP                 2      2      993        0  0.0000s
...

8. Dshell Plugin Packs

As an option for distribution and keeping custom plugins separate from those native to the framework, groups of custom plugins can be organized and installed as a plugin pack. Updates to the Dshell Python package will overwrite plugins stored in the Dshell installation directories: [...]/site-packages/dshell/ and [...]/Dshell/dshell/plugins but will not overwrite plugin packs. A pack is configured and built using the setuptools Python module and defining plugins as entry points.

When developing a setup.py script for a plugin pack, it is necessary to define a "dshell_plugins" key in the entry_points argument dictionary. Dshell’s decode.py checks this entry point key for plugins and adds them to the list of available plugins. Additionally, the install_requires argument should include “Dshell.”

For example, imagine a project of custom plugins. The project is arranged with a top-level directory, a setup.py script, and a subdirectory containing the plugins example.py and test.py:

```
Project/
    Project/setup.py
    Project/example_plugins/
        Project/example_plugins/example.py
        Project/example_plugins/test.py
```

The following script is an example of a setup.py that can package the custom plugins into a plugin pack accessible by Dshell. It provides a name and other metadata for the plugin pack, lists “Dshell” as an installation requirement, and includes the “example,” “referer,” “rot13,” and “netflow-ct” import paths in the "dshell_plugins" entry_point key.
# setup.py

```python
from setuptools import find_packages, setup

setup(
    name="Dshell-Example-Pack",
    version="0.1",
    author="USArmyResearchLab",
    description="A collection of Dshell plugins used for example purposes",
    url="https://github.com/USArmyResearchLab/Dshell",
    python_requires='>=3.8',
    packages=find_packages(),
    install_requires=[
        "Dshell",
    ],
    entry_points={
        "dshell_plugins": [
            "example = example_plugins.example",
            "referer = example_plugins.referer",
            "rot13   = example_plugins.rot13",
            "netflow_ct = example_plugins.netflow_ct",
        ],
    },
)
```

With the setup.py script, the plugin pack can be installed directly from the project directory using Python’s package installer pip: `pip3 install`.

Alternatively, the plugin pack can be packaged for distribution using setup.py directly. Python’s setuptools provides many options for creating a distribution package, but a general command line call would look like this: `python3 setup.py sdist`. Following the creation of the package, usually stored in the project’s dist directory, it can be installed with pip: `pip3 install [package file].`

An installed plugin pack can be removed using a standard pip command: `pip3 uninstall [package file].`
9. References


<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>ARL</td>
<td>Army Research Laboratory</td>
</tr>
<tr>
<td>ASN</td>
<td>autonomous system number</td>
</tr>
<tr>
<td>BPF</td>
<td>Berkeley Packet Filter</td>
</tr>
<tr>
<td>C2</td>
<td>command and control</td>
</tr>
<tr>
<td>CLI</td>
<td>command-line interface</td>
</tr>
<tr>
<td>DEVCOM</td>
<td>US Army Combat Capabilities Development Command</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>Dshell</td>
<td>decoder-shell</td>
</tr>
<tr>
<td>HTTP</td>
<td>hypertext transfer protocol</td>
</tr>
<tr>
<td>ID</td>
<td>identification</td>
</tr>
<tr>
<td>IP</td>
<td>Internet protocol</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>TCP</td>
<td>transport control protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>user datagram protocol</td>
</tr>
<tr>
<td>URI</td>
<td>uniform resource identifier</td>
</tr>
<tr>
<td>VLAN</td>
<td>virtual local area network</td>
</tr>
</tbody>
</table>