# Packet Debugger

George V. Neville-Neil

January 19, 2007

## 1 Introduction

The Packet Debugger, *pdb* is a program which allows people to work with packet streams as if they were working with a source code debugger. Users can list, inspect, modify, and retransmit any packet from captured files as well as work with live packet capture.

Installing `pdb` is covered in the text file, `INSTALLATION`, which came with this package. The code is under a BSD License and can be found in the file `COPYRIGHT` in the root of this package.

*Note: You will very likely need root or sudo access in order to write packets directly to a network interface, or read them directly from it. If you don't understand this note, then please talk to your local systems or network administrator before trying to use `pdb` to read and write raw packets.*

## 2 A Quick Tour

For the impatient this section is a 5 minute intro to using the packet debugger.

Create a `pcap` file with `tcpdump`, `ethereal`, `wireshark` or another program of your choosing. Now load the `pcap` file into `pdb` as shown in Figure 1. Figure 1 will serve as our only figure throughout this section. You said you were impatient, didn't you?

The first thing to do when you start a new program is to ask for help, and `pdb` is no different in this respect. The complete command set is described in the built in help system. You can ask for help on each command as well, but that is not shown in this section.

`pdb` attempts to at very much like a well known debugger and so, if you're a programmer, you're very likely to recognize many of the commands.

```
minion ? sudo src/pdb.py -f tests/test.out -i en0
Welcome to PDB version Alpha 0.1.
For a list of commands type 'help <rtn>'
For help on a command type 'help command <rtn>'
pdb> help

Documented commands (type help <topic>):
========================================
break     delete  list  next  print  run   set   unload
continue  info    load  prev  quit   send  show

pdb> print

0: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048,
        sha: '\x00\r\x022{\x9c', pln: 4, hrd: 1,
        tha: '\x00\x00\x00\x00\x00\x00', op: 1>

pdb> list
0: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048, sha: '\x00\r\x022{\x9c

1: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048, sha: '\x00\r\x022{\x9c

2: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>  <IP
  <TCP: reset: 6, reserved: 0, sequence: 3630104920L, ack: 1, checksum: 1430, offset:
  <Data: payload: 13461827120112604152439673499521091613012183756744016516126079577 20

3: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\x00\x17\xf2\xe8\x9a*', type: 2048>  <IP
  <TCP: reset: 6, reserved: 0, sequence: 4015249839L, ack: 1, checksum: 64954, offset
  <Data: payload: 583301264018269383074068530912649179318369935407230015125988744911 1

[NOTE: Packets 4 through 9 removed for brevity]

pdb> run
pdb> quit
Bye
```

Figure 1: Quick starting pdb

```
usage: pdb.py [options]

options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        pcap file to read
  -i INTERFACE, --interface=INTERFACE
                        Network interface to connect to.
```

Figure 2: `pdb` command line arguments

In our example we've loaded the test data used to test this program, `test.out`. Each file or set of packets is part of a stream, and in this example we have one stream, which was loaded from `test.out`. We are currently at position 0 in the stream, the beginning. We can print the packet at the current position with the `print` command, as shown in the example. What we see is an Ethernet packet, containing an ARP request. We can also list all the packets in the stream, up to a user configured limit. The `list` command shows, by default, 10 packets, including the one at your current position and the following 9. To play back a stream over the interface selected at startup the `run` command is used. If you pick an Ethernet interface at startup, as we did with `en0`, then the packet stream will be sent over that interface. To see the packets you're playing back you can run `tcpdump` or a similar packet capture program, to see the packets coming from `pdb`.

## 3 Starting `pdb`

In order to start a debugging session you will need either a pre-recorded `pcap` file or a network interface to work with, and possibly both. The command line arguments to `pdb` are relatively simple and are shown in Figure 2.

The `-f` or `--file` switch supplies a path and file name to `pdb` which it will then attempt to load into the program. If no `-i` or `--interface` argument is supplied then the user can only read packets from the file. Other files and interfaces may be opened from the command line, see Sections

Once `pdb` has started you will see the command prompt, shown in Figure 3.

At this point `pdb` is awaiting your commands.

```
pdb>
```

Figure 3: Command Prompt

| | |
|---|---|
| Ctrl-b | Back up one character |
| Ctrl-f | Move forward one character |
| Ctrl-a | Move to the beginning of the line |
| Ctrl-e | Move to the end of the line |
| Enter | Ask `pdb` to execute the command |
| Tab | Complete command |

Table 1: CLI Editing Keys

### 3.1 Working with the Command Line

The Command Line Interpreter (CLI) in `pdb` is implemented using the CMD module in Python, which in turn uses the popular `readline` package. What all of that means is that you have fairly rich, built in command line functions, including the ability to repeat, edit, and complete command lines. We are not going to reproduce all of the documentation on `readline` in this section but will give a brief introduction to what the CLI provides. If you have worked with any modern Unix shell, i.e. bash, tcsh, etc., you will be quite comfortable using the `pdb` CLI.

As with any other command line your cursor waits at the prompt for your input. You can ask for `help` which will give you a list of commands to choose from, and you can ask for help on a particular command, which will explain the command itself. When you are entering characters on the command line you can use a few special keys to edit the text you have already entered, and these keys are listed in Table 1.

Command completion is the ability of the CLI to guess, based on a few characters, what command you're trying to give to it. Using the Tab key frequently is a good way to avoid typing too much or making typing mistakes. If the CLI is unable to undestand the command you're trying to complete it will tell you, by either going no further in the command line when you type Tab, or by giving you a set of choices of possible commands to complete. Pressing Tab when there is no text after the command prompt will give you a list of all the available commands. Some commands also have completion based on the data you are trying to work with, such as a list of streams, and these special cases are covered in sections 4.5.1, 4.4.1, 4.4.3, and 4.7.2, which cover the commands that have completion.

Unlike a Unix shell exiting by the Ctrl-d (EOF) key is not supported, though the program can be halted using Ctrl-c. We strongly recommend using the `quit` command to exit the program.

# 4 Command Reference

All of the commands implemented in `pdb` are covered in this section and its subsequent sub-sections.

## 4.1 help

The help command prints out the available topics for help.

```
pdb> help

Documented commands (type help <topic>):
========================================
break     delete  info  load  prev   quit  send  show
continue  help    list  next  print  run   set   unload
```

Figure 4: Help on all commands

To get help on a specific command type `help command` where `command` is one of the commands listed when you ask for help on its own.

```
pdb> help help
help [command]
print out the help message, with [command] get help on that comamnd
```

Figure 5: Help on the help command

## 4.2 quit

Quit the program. All program state is lost. In the next version it will be possible to save the state of your streams before exiting.

## 4.3 Loading and Saving Streams

Each of the commands in this section works on a stream, which is the basic unit on which pdb operates.

```
pdb> quit
Bye
localhost ?
```

Figure 6: Quit Command

#### 4.3.1 load

Read a new stream from a file, or open a network connection. Currently only `pcap` files are supported by the `load` command.

```
pdb> load filename tests/test.out
```

Figure 7: Load example

#### 4.3.2 unload

Unload a previously loaded stream. If a numeric argument is supplied then `pdb` will attempt to unload that stream. To see all the currently loaded streams use the `info` command, discussed in Section 4.4.1.

```
pdb> unload
```

Figure 8: Unload command

### 4.4 Inspecting a Stream

Once a STREAM is loaded into `pdb` you will want to work with it in various ways. In this section we cover all the commands that allow you to inspect and move through a STREAM.

#### 4.4.1 info

Get information on all the streams currently loaded into `pdb`. The stream displayed in Figure 9 has no breakpoints, was loaded from one of our standard test files, `tests/test.out`, has no filter set, and contains 63 packets. We are currently at the first packet, position 0, in the stream. The `Type` is not yet supported. The stream is an ISO Layer 2 stream, with a `Datalink`

type of Ethernet, which has a 14 byte offset between the link layer header and the next protocol.

```
Stream 0
---------
Breakpoints []
File tests/test.out
Filter
Number of packets: 63
Current Position: 0
Type
Layer: 2
Datalink: 1     Offset: 14
```

Figure 9: Getting info on a STREAM

#### 4.4.2 print

The `print` command is used to show a single packet, either the current packet or another one in the same stream.

```
0: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048,
        sha: '\x00\r\x022{\x9c', pln: 4, hrd: 1,
        tha: '\x00\x00\x00\x00\x00\x00', op: 1>
```

Figure 10: print command

Figure 10 shows the first, 0th, packet in one our supplied test files, `test.out`. Packet 0 is an Ethernet frame containing an ARP request.

The `print` command gives a concise example of how packets are displayed by `pdb`. Packets are displayed from the lowest available layer, upwards towards the highest available layer, as viewed using the ISO standard for networking. Ethernet is the lowest layer we have captured, and the only other data we have is the ARP packet placed, logically speaking, on top of it. Each layer is displayed on its own line.

With in each packet the fields are given somewhat human readable names, that is, if the human is acquainted with network protocols. Most of

pdb assumes that the user has at least a passing understanding of networking and the ability to look up information about packet formats and field names on their own.

### 4.4.3 list

The list command shows a subset of the packets in a stream. The number of packets shown is controlled by the list_length setting, see Section 4.7.2, which defaults to 10.

```
pdb> list
0: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048, sha: '\x00\r\x022{\x9c

1: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048, sha: '\x00\r\x022{\x9c

2: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>  <IP
  <TCP: reset: 6, reserved: 0, sequence: 3630104920L, ack: 1, checksum: 1430, offset:
  <Data: payload: 13461827120112604152439673499521091613012183756744016516126079577 20
```

Figure 11: The list command

In Figure 11 we see a subset of the packets printed by the list command. Each packet is represented just as it is with the print command, explained in Section 4.4.2.

### 4.4.4 next

To move within a Stream there are two commands provided, the next command moves you forward, while the prev command, Section 4.4.5 moves you backwards. An optional numeric argument can be given to move more than 1 packet at a time. As we see in Figure 12 each time you use the next command the packet you have jumped to is printed for you, to let you know where you are.

We were originally at packet number 1, shown by the print command, and then after the next command we are at packet number 2. Attempts to jump past the end or beginning of the stream are reported as errors, and no change is made to your position in the Stream.

```
pdb> print
1: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
  <ARP: spa: 3538819329L, tpa: 3538819566L, hln: 6, pro: 2048, sha: '\x00\r\x022{\x9c
pdb> next
2: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>  <IP
  <TCP: reset: 6, reserved: 0, sequence: 3630104920L, ack: 1, checksum: 1430, offset:
  <Data: payload: 134618271201126041524396734995210916130121837567440165161260795772O
```

Figure 12: The next command

#### 4.4.5 prev

To move within a STREAM there are two commands provided, the `prev` command moves you backwards, while the `next` command, Section 4.4.4 moves you forwards. An optional numeric argument can be given to move more than 1 packet at a time. Please refer to Section 4.4.4 for more information.

### 4.5 Running a Stream

Once a stream is loaded or captured you may want to replay the stream on an interface. In almost all cases playing raw packets back on an interface requires special privileges, usually those associated with the `root` user. On modern Unix systems (FreeBSD, NetBSD, OpenBSD, MacOS X, Linux, Solaris, etc.) the best way to gain this privilege is via the `sudo` command. If you do not understand what was just explained here, please stop, and find someone to explain it to you.

#### 4.5.1 run

The `run` command is used to play a stream of packets on an interface. To use a network interface it must be specified when `pdb` is started, see Section 4.7, and at the moment the network interface used for output must match the type of interface on which the packets were captured. A stream of packets captured on an Ethernet interface *must* be run on an Ethernet interface and a stream of packets captures on the localhost, `lo0`, interface *must* be played back on the localhost interface.

There is no output from the `run` command to the CLI. When playback is complete the command line returns, as seen in Figure 13.

```
pdb> run
pdb>
```

Figure 13: The run command

### 4.5.2 break

One of the main features of any debugger is to be able to stop a program at a specific point in its execution. Such a point is called a breakpoint and the `break` command is used to set a breakpoint in a STREAM. Since `pdb` works with streams of packets, and not lines of source code, the breakpoints are set on packets, and not source code lines.

The `break` command sets a break point at a particular packet so that when the stream is `run`, `pdb` will send packets up to the breakpoint, and then stop, returning control to the user at the command line.

In Figure14 we have set a breakpoint at packet number 5, and then run the stream using the `run` command. Just before `pdb` is about to transmit packet number 5 it stops, and returns control to the user. The user can now inspect the packet, wait for an event in their program, or do something else with `pdb`.

```
pdb> break 5
pdb> run
Breakpoint at packet 5
5: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>  <IP
  <TCP: reset: 6, reserved: 0, sequence: 3630104973L, ack: 1, checksum: 26250, offset
  <Data: payload: 134618271201126041524525709191870063484130928604002291753553763921 8
pdb>
```

Figure 14: The break command

### 4.5.3 continue

When `pdb` reaches a breakpoint, see Section 4.5.2, it halts transmitting the packet stream. If the user were to give the `run` command again `pdb` would start transmitting packets from the 0th packet and then reach the same breakpoint again. The `continue` command continues transmitting packets from the stream from the current point, the one it reached when it hit the breakpoint. With the `continue` command it is possible to set and

reach several breakpoints and to then move consistently through the packet stream.

## 4.6 Working with Packets

In the previous sections we were working with streams of packets, but not with individual packets themselves.

### 4.6.1 send

The `send` command is used to send a single packet from the current stream. When used without any arguments it sends the packet at the current position. With a numeric argument it sends the packet at the numbered index in the stream. No output is shown in the CLI when this command is used.

### 4.6.2 delete

The `delete` command is used to remove a packet from the packet stream. When used without any arguments it deletes the packet at the current position. With a numeric argument deletes the packet at the numbered index in the stream. No output is shown in the CLI when this command is used.

## 4.7 Debugger Options

Various options may be globally set for the packet debugger. The `show` and `set` commands allow the user to see the options and to modify them.

### 4.7.1 show

The `show` command lists the values of all the possible packet debugger options. Currently there are only two options, `list_length` and `layer`. The `list_length` option controls how many packets are displayed when the user invokes the `list` command, See Section 4.4.3.

```
pdb> show
list_length = 10
layer = -1
```

Figure 15: Global Debugger Options

### 4.7.2 set

The `layer` option restricts packet output to a specific ISO layer. The default value, -1, shows all layers simultaneously. If the user wants to only inspect a particular layer of packets they can set this to any value from 1 through 7. Figure 16 shows an example of output restricted to the data-link layer, in this case, Ethernet.

```
pdb> set layer 2
pdb> list
0: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
1: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\xff\xff\xff\xff\xff\xff', type: 2054>
2: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>
3: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\x00\x17\xf2\xe8\x9a*', type: 2048>
4: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>
5: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>
6: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\x00\x17\xf2\xe8\x9a*', type: 2048>
7: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>
8: <Ethernet: src: '\x00\x17\xf2\xe8\x9a*', dst: '\x00\r\x022{\x9c', type: 2048>
9: <Ethernet: src: '\x00\r\x022{\x9c', dst: '\x00\x17\xf2\xe8\x9a*', type: 2048>
```

Figure 16: Output restricted to Layer 2, datalink