

Name: Archit Benipal

Subject: Software Supply Chain Security (CY 653)

Assignment 9: Other BCC tools

Biolatency BCC tool:

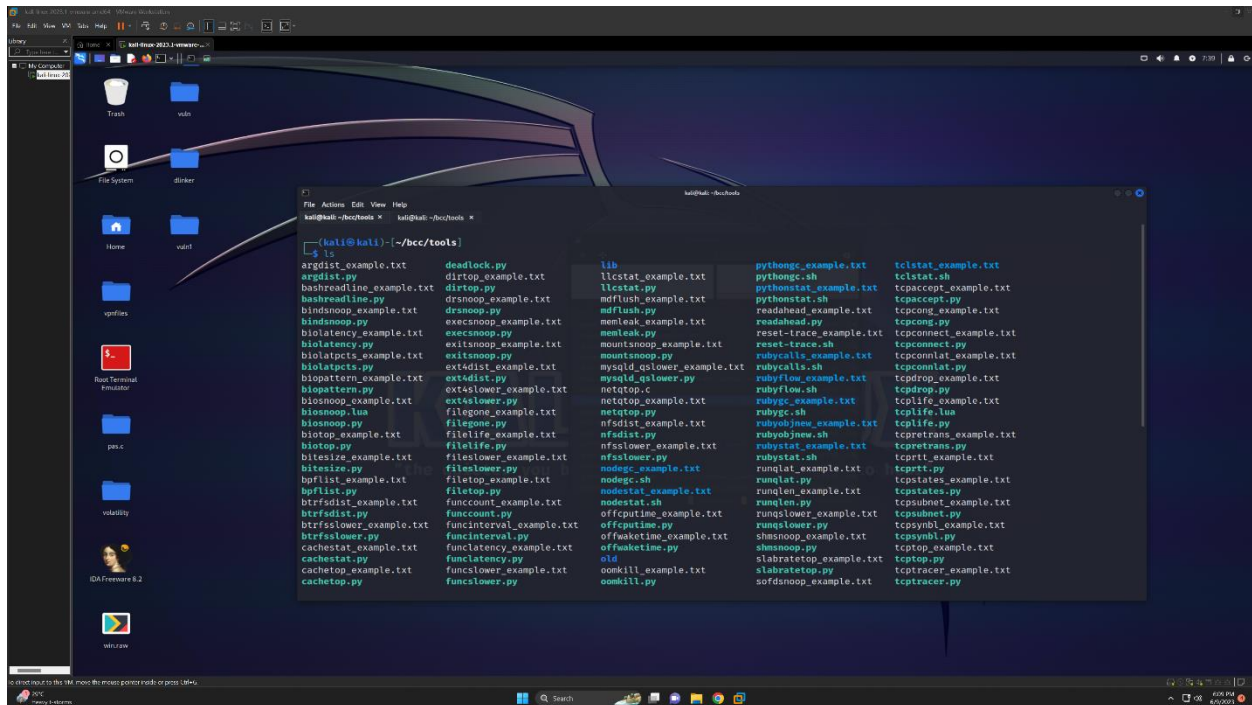
What is biolateness:

It is a tool included in the BCC (BPF Compiler Collection) that uses eBPF (Extended Berkeley Packet Filter) to track the completion time of block device I/O (input/output) operations, often referred to as block I/O or simply bio. It effectively measures the latency of disk I/O operations in a Linux system.

When it is executed, it creates a histogram showing the distribution of I/O latency in milliseconds. This can be useful for identifying whether your storage subsystem is experiencing delays. It might help in diagnosing storage performance issues and can be part of a comprehensive performance analysis methodology.

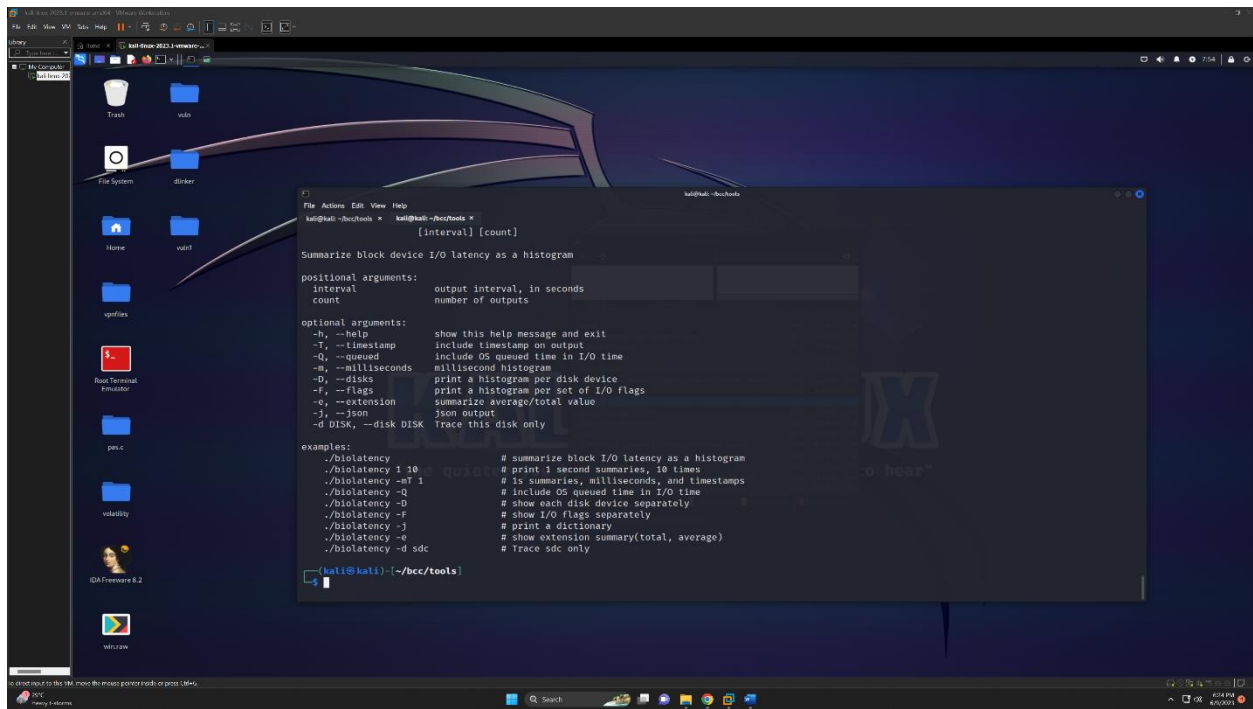
Usage:

Step 1: Cloned the Github repo of the bcc tools and navigated to the /bcc/tools directory to list all the tools



Here we can see list of several tools with the instruction manual for the usage of each tool and access to the old tools available in the previous versions of BCC.

Step 2: Examples



The screenshot shows a Kali Linux desktop environment. A terminal window is open, displaying the help output for the `biolatility` script. The output includes positional arguments (`interval` and `count`), optional arguments (`-h`, `-T`, `-Q`, `-m`, `-D`, `-f`, `-e`, `-j`, `-d`), and examples of how to use the script.

```
File Actions Edit View Help
kali@kali:~/bcc/tools$ ./biolatility.py [interval] [count]

Summarize block device I/O latency as a histogram

positional arguments:
  interval              output interval, in seconds
  count                number of outputs

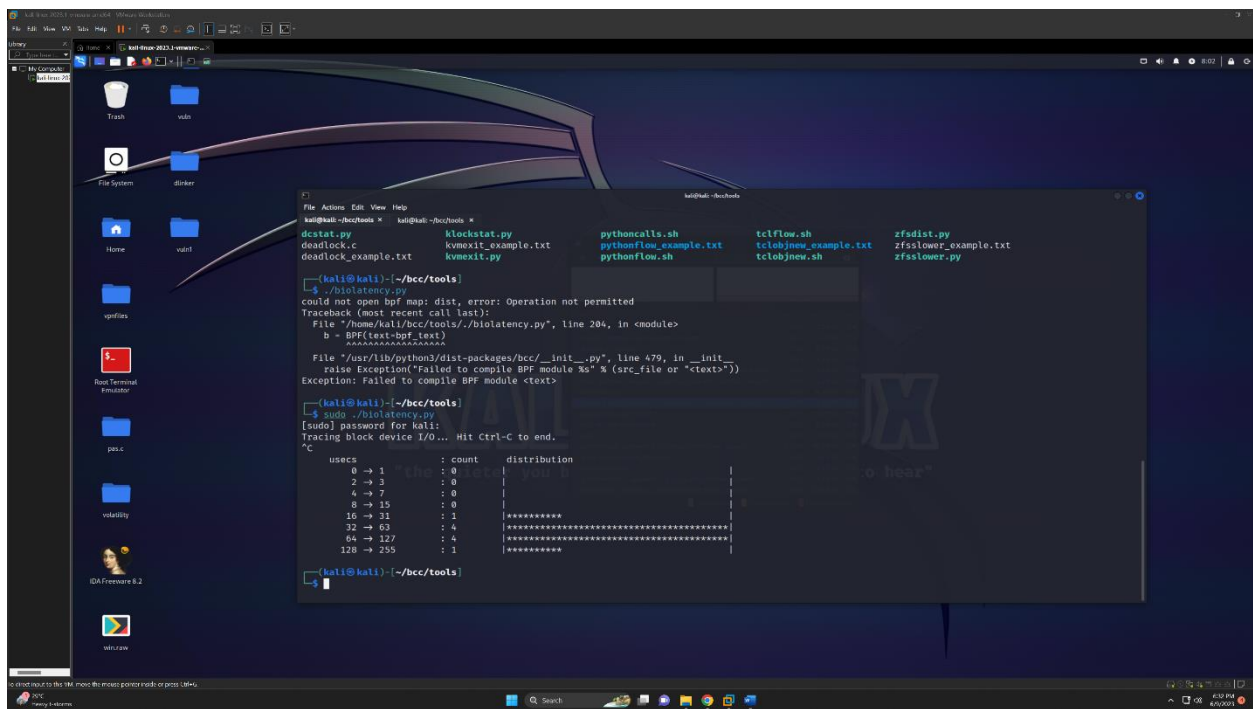
optional arguments:
  -h, --help            show this help message and exit
  -T, --timestamp       include timestamp on output
  -Q, --queued          include OS queued time in I/O time
  -m, --milliseconds   millisecond histogram
  -D, --disks           print a histogram per disk device
  -f, --flags           print a histogram per set of I/O flags
  -e, --extension       summarize average/total value
  -j, --json            json output
  -d DISK, --disk DISK Trace this disk only

examples:
  ./biolatility.py                # summarize block I/O latency as a histogram
  ./biolatility.py 1 10          # print 1 second summaries, 10 times
  ./biolatility.py -T 1          # 1s summaries, milliseconds, and timestamps
  ./biolatility.py -Q            # include OS queued time in I/O time
  ./biolatility.py -D            # show each disk device separately
  ./biolatility.py -f            # show I/O flags separately
  ./biolatility.py -j            # print a dictionary
  ./biolatility.py -e            # show extension summary(total, average)
  ./biolatility.py -d sdc        # Trace sdc only

kali@kali:~/bcc/tools$
```

Here we can see all the examples with this script.

Step 3: Executed the biolatility script with no switches



The screenshot shows the same Kali Linux desktop environment. The terminal window now shows the execution of the `biolatility.py` script with no switches. It displays a traceback error, followed by a successful execution of the script, which produces a histogram of block device I/O latency.

```
File Actions Edit View Help
kali@kali:~/bcc/tools$ ./biolatility.py
could not open bpf map: dist, error: Operation not permitted
Traceback (most recent call last):
  File "/home/kali/bcc/tools/./biolatility.py", line 204, in <module>
    b = BPF(text=bpf_text)
  File "/usr/lib/python3/dist-packages/bcc/_init_.py", line 479, in __init__
    raise Exception("Failed to compile BPF module %s" % (src_file or "c/text"))
Exception: Failed to compile BPF module <text>

kali@kali:~/bcc/tools$ ./biolatility.py
[sudo] password for kali:
Tracing block device I/O... Hit Ctrl-C to end.
^C

users      : count  distribution
0 -> 1      : 0      |
2 -> 3      : 0      |
4 -> 7      : 0      |
8 -> 15     : 0      |
16 -> 31    : 1      |
32 -> 63    : 4      |
64 -> 127   : 4      |
128 -> 255  : 1      |

kali@kali:~/bcc/tools$
```

Breakdown:

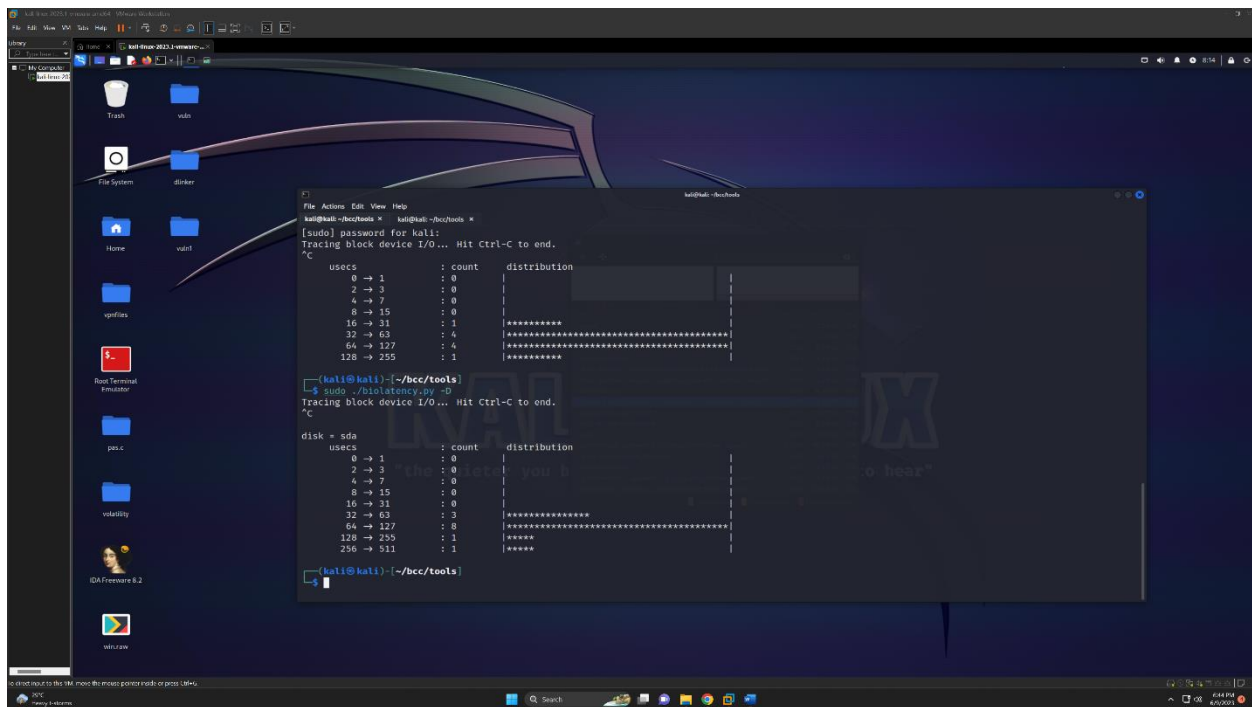
The script traces block device I/O latency

- The script attaches to the block device I/O tracing mechanism and starts collecting data on the latency of I/O operations.
- The output displays a histogram-like representation of the I/O latency distribution.
- The column labeled "usecs" represents the different ranges of I/O latency in microseconds.
- The column labeled "count" shows the number of I/O operations that fall within each latency range.
- The bar graph represents the distribution of I/O operations across the latency ranges, with the "*" character used to fill the bars.
- The longer the bar, the higher the count of I/O operations within that particular latency range.

Here it can also be observed that

- There were no I/O operations with a latency in the range of 0 to 7 microseconds.
- There was one I/O operation with a latency in the range of 16 to 31 microseconds.
- There were four I/O operations each with latencies in the ranges of 32 to 63 microseconds and 64 to 127 microseconds.
- There was one I/O operation with a latency in the range of 128 to 255 microseconds.

Step 4: Ran the script with the -D switch to show each disk separately



```
File Actions Edit View Help
kali@kali:~/hcc/tools$ ./disklatency.py -D
[sudo] password for kali:
Tracing block device I/O... Hit Ctrl-C to end.
^C
  usecs      : count  distribution
  0 -> 1      : 0      |
  2 -> 3      : 0      |
  4 -> 7      : 0      |
  8 -> 15     : 0      |
  16 -> 31    : 1      | *****
  32 -> 63    : 4      | *****
  64 -> 127   : 4      | *****
  128 -> 255  : 1      | *****

[kali@kali:~/hcc/tools]$
kali@kali:~/hcc/tools$ ./disklatency.py -D
Tracing block device I/O... Hit Ctrl-C to end.
^C
disk = sda
  usecs      : count  distribution
  0 -> 1      : 0      |
  2 -> 3      : 0      |
  4 -> 7      : 0      |
  8 -> 15     : 0      |
  16 -> 31    : 0      |
  32 -> 63    : 3      | *****
  64 -> 127   : 8      | *****
  128 -> 255  : 1      | *****
  256 -> 511  : 1      | *****

[kali@kali:~/hcc/tools]$
```

Breakdown:

As we have allotted only one disk to the VM we can see only one disk in the output

Here it can be Observed that:

- There were no I/O operations with a latency in the range of 0 to 15 microseconds.
- There were three I/O operations with latencies in the range of 32 to 63 microseconds.
- There were eight I/O operations with latencies in the range of 64 to 127 microseconds.
- There was one I/O operation with a latency in the range of 128 to 255 microseconds.
- There was one I/O operation with a latency in the range of 256 to 511 microseconds.

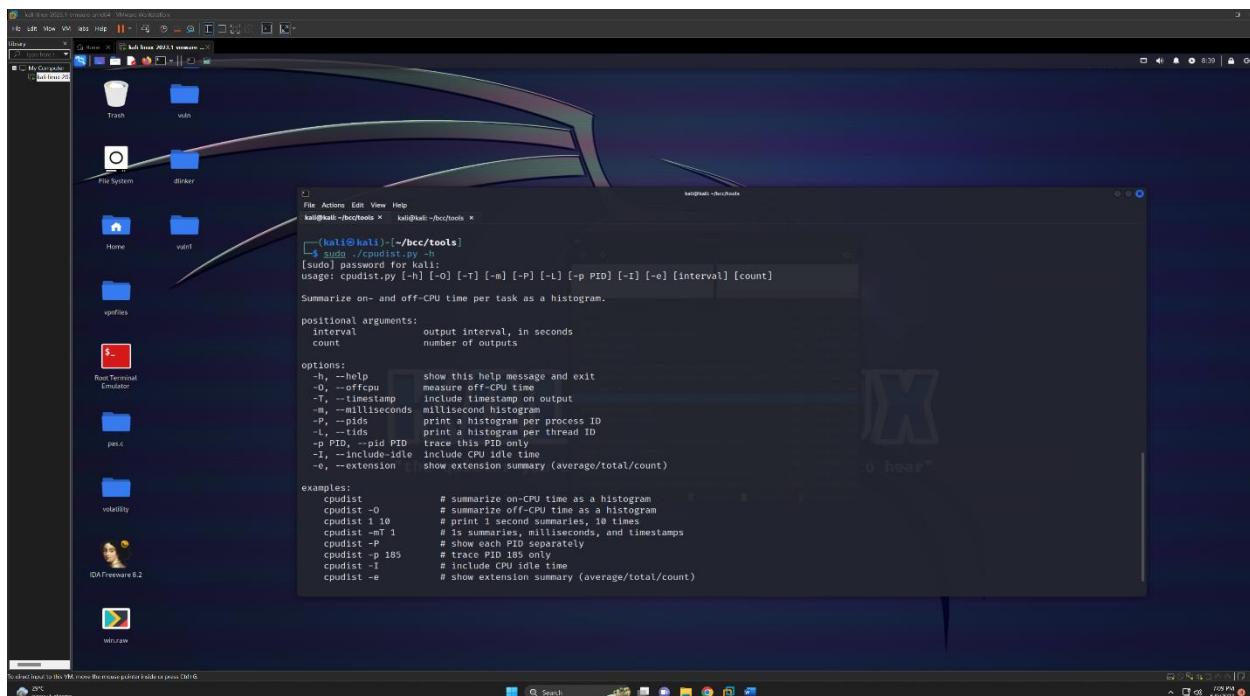
CPUdust BCC tool:

What is CPUdust

It is a BCC-based tool that uses eBPF (Extended Berkeley Packet Filter) to measure CPU distribution in a Linux system. More specifically, **cpudist** measures the distribution of CPU time spent by threads while they are running on a CPU. This includes only the time spent running on a CPU, not while waiting in run queues. It creates a histogram showing the distribution of time spent on CPU. This information can be useful for identifying whether there are any performance issues related to CPU usage. The output might assist in diagnosing CPU-bound applications or identifying CPU performance anomalies.

Usage:

Step 1: Ran the `--help` command on `cpudist.py`



```
kali@kali:~/bcc/tools$ sudo ./cpudist.py --help
[sudo] password for kali:
usage: cpudist.py [-h] [-O] [-T] [-m] [-P] [-L] [-p PID] [-i] [-e] [interval] [count]

Summarize on- and off-CPU time per task as a histogram.

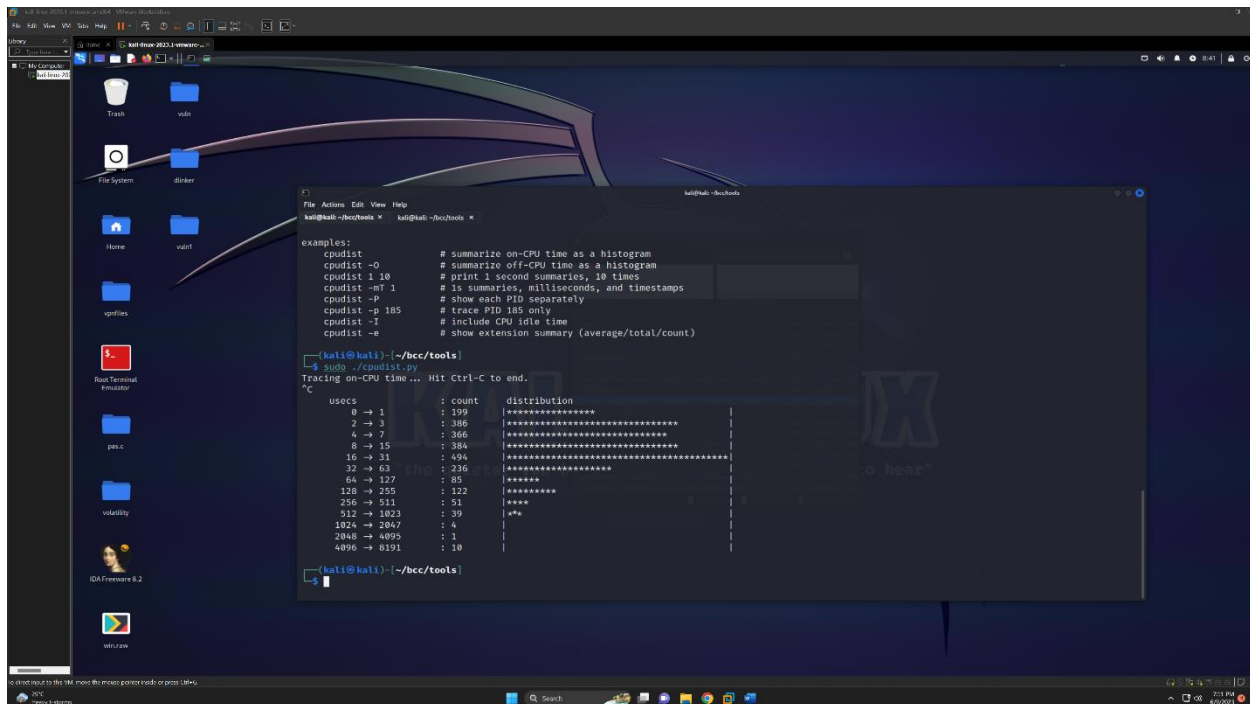
positional arguments:
  interval              output interval, in seconds
  count                number of outputs

options:
  -h, --help            show this help message and exit
  -O, --offcpu          measure off-CPU time
  -T, --timestamp       include timestamp on output
  -m, --milliseconds    millisecond histogram
  -P, --pids            print a histogram per process ID
  -L, --tids            print a histogram per thread ID
  -p PID, --pid PID     trace this PID only
  -I, --include-idle    include CPU idle time
  -e, --extension       show extension summary (average/total/count)

examples:
cpudist                # summarize on-CPU time as a histogram
cpudist -O             # summarize off-CPU time as a histogram
cpudist 1 10           # print 1 second summaries, 10 times
cpudist -mT 1         # 1s summaries, milliseconds, and timestamps
cpudist -p            # show each PID separately
cpudist -p 185        # trace PID 185 only
cpudist -I            # include CPU idle time
cpudist -e            # show extension summary (average/total/count)
```

Here we can see the basic examples and the usage of cpudist.py script

Step 2: Executing the cpudist.py



Breakdown:

This script traces the on-CPU time, which refers to the amount of time that a process or thread spends executing on the CPU.

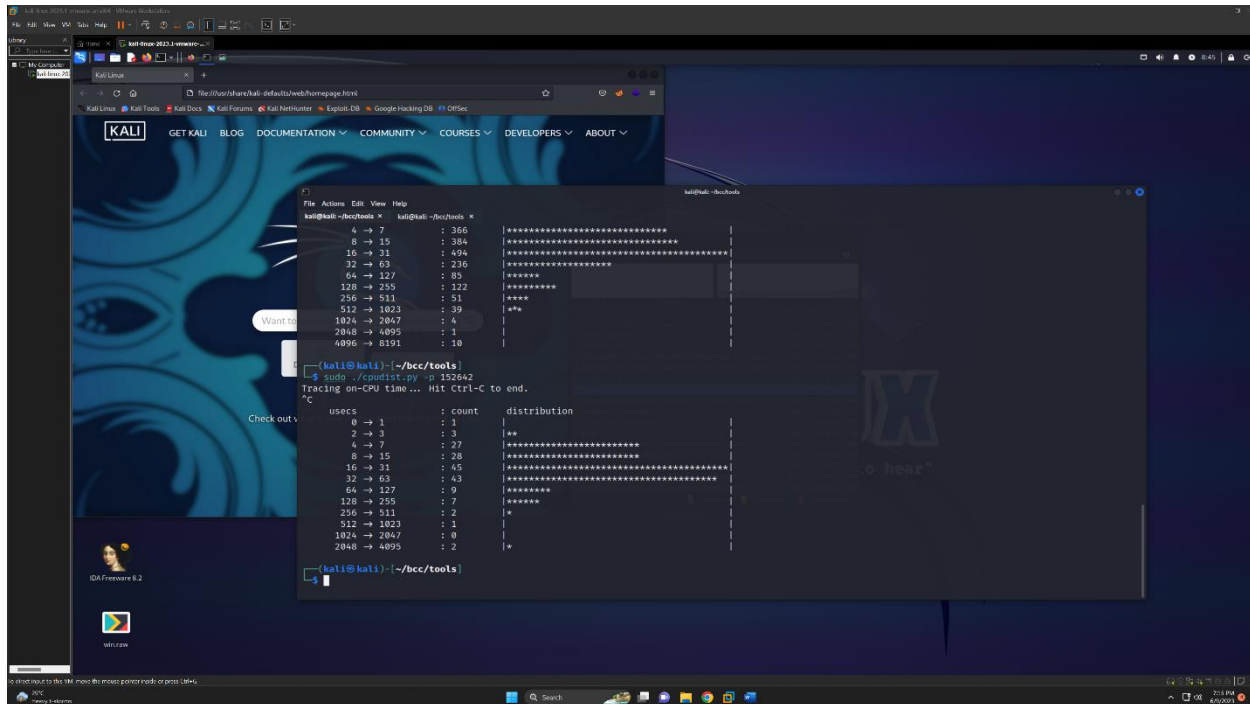
- The column labeled "usecs" represents the different ranges of on-CPU time in microseconds.
- The column labeled "count" shows the number of events (in this case, the number of times a process or thread had a particular on-CPU time) that fall within each time range.
- The bar graph represents the distribution of events across the time ranges, with the "*" character used to fill the bars.

It can also be observed that:

- There were 199 events with on-CPU time in the range of 0 to 1 microseconds.
- There were 386 events with on-CPU time in the range of 2 to 3 microseconds.
- There were 366 events with on-CPU time in the range of 4 to 7 microseconds.
- There were 384 events with on-CPU time in the range of 8 to 15 microseconds.
- There were 494 events with on-CPU time in the range of 16 to 31 microseconds.
- There were 236 events with on-CPU time in the range of 32 to 63 microseconds.
- There were 85 events with on-CPU time in the range of 64 to 127 microseconds.
- There were 122 events with on-CPU time in the range of 128 to 255 microseconds.
- There were 51 events with on-CPU time in the range of 256 to 511 microseconds.
- There were 39 events with on-CPU time in the range of 512 to 1023 microseconds.
- There were 4 events with on-CPU time in the range of 1024 to 2047 microseconds.

- There was 1 event with on-CPU time in the range of 2048 to 4095 microseconds.
- There were 10 events with on-CPU time in the range of 4096 to 8191 microseconds.

Step 3: Tracing a particular PID



Breakdown:

Here we launched firefox and tried to trace the PID , It can also be observed that

- There was 1 event with on-CPU time in the range of 0 to 1 microsecond.
- There were 3 events with on-CPU time in the range of 2 to 3 microseconds.
- There were 27 events with on-CPU time in the range of 4 to 7 microseconds.
- There were 28 events with on-CPU time in the range of 8 to 15 microseconds.
- There were 45 events with on-CPU time in the range of 16 to 31 microseconds.
- There were 43 events with on-CPU time in the range of 32 to 63 microseconds.
- There were 9 events with on-CPU time in the range of 64 to 127 microseconds.
- There were 7 events with on-CPU time in the range of 128 to 255 microseconds.
- There were 2 events with on-CPU time in the range of 256 to 511 microseconds.
- There was 1 event with on-CPU time in the range of 512 to 1023 microseconds.
- There were 0 events with on-CPU time in the range of 1024 to 2047 microseconds.
- There were 2 events with on-CPU time in the range of 2048 to 4095 microseconds.

Step 4: View each PID separately

```
File Actions Edit View Help
kali@kali:~/tools$ sudo ./cpudist.py -P
Tracing on-CPU time... Hit Ctrl-C to end.
^C

pid = 57896 xdg-desktop-por
      ussecs      : count      : distribution
      0 -> 1      : 0        : |
      2 -> 3      : 0        : |
      4 -> 7      : 0        : |
      8 -> 15     : 0        : |
      16 -> 31    : 0        : |
      32 -> 63    : 1        : |*****|

pid = 1519 nm-applet
      ussecs      : count      : distribution
      0 -> 1      : 0        : |
      2 -> 3      : 0        : |
      4 -> 7      : 0        : |
      8 -> 15     : 0        : |
      16 -> 31    : 0        : |
      32 -> 63    : 0        : |
      64 -> 127   : 1        : |*****|

pid = 48 migration/5
      ussecs      : count      : distribution
      0 -> 1      : 0        : |
      2 -> 3      : 1        : |*****|

pid = 773 Xorg
```

Command used “sudo ./cpudist.py -P”

Breakdown:

For process with PID 57896 (**xdg-desktop-por**):

- There were no events with on-CPU time in the range of 0 to 63 microseconds.
- There was 1 event with on-CPU time in the range of 32 to 63 microseconds.

For process with PID 1519 (**nm-applet**):

- There were no events with on-CPU time in the range of 0 to 63 microseconds.
- There was 1 event with on-CPU time in the range of 64 to 127 microseconds.

Team Name:

- ALL Safe

Members:

- Archit Benipal
- Siva Prasad Kolli
- Venkata Nethaji Yenduri