SANS DFIR JSON and jq Quick Start Guide v1.2

This guide was created by Phil Hagen & David Szili — lewestech.com | alzetteinfosec.com

Purpose

This guide is a supplement to the SANS **FOR572: Advanced Network Forensics: Threat Hunting, Analysis, and Incident Response** course. It covers the basics of JSON and some of the fundamentals of the **jq** utility. The **jq** utility filters, parses, formats, and restructures JSON—think of it as **sed**, **awk**, and **grep**, but for JSON. Given the trend toward logs being generated in JSON, easily accessing and molding that data is increasingly important for the forensicator. This document is not intended to replace **jq**'s extensive documentation. It is only a quick reference resource.

Acquiring the jq Utility

Stephan Dolan is the developer of **jq**, which is free and open-source software. You can install **jq** in most *NIX-based operating systems using the distribution's software/package management system, or you can download the binary for your operating system or the source code from the main project page. Visit **for572.com/jq** for details, as well as to learn how to contribute to the project through the author's GitHub repository.

A web-based version of **jq** is hosted at **jqplay.org**. This is especially helpful while learning **jq**, as you can experiment with filters and options in a graphical interface. The **jqplay.org** web implementation can also be hosted in your own environment, making it suitable for air-gapped networks.

JSON: Structure for Humans and Machines

JSON (JavaScript Object Notation) is an open standard data interchange format, which is both machine-parsable and (mostly) human-readable. JSON is built on two primary structures: Key/value pairs and ordered lists of values. Data types for values can consist of strings, numbers, booleans, nested JSON objects, or the null value.

A single-field JSON object might look like this:

```
{ "name": "Lance" }
```

A more complex JSON object might look like this:

```
{ "name": "Lance", "age": 42,
  "active": true, "tags": [ "tag1",
  "tag2" ], "address": { "street":
  "123 Main", "city": "Lewes",
  "state": "DE", "postalCode":
  "19958"}, "pet": null }
```

JSON can be represented in compact form, as shown above with one object per line, or expanded as shown below – both formats are considered equivalent. Note that **tags** is an array and **address** is a nested JSON object.

```
{
  "name": "Lance",
  "age": 42,
  "active": true,
  "tags": [
    "tag1",
    "tag2"
  ],
  "address": {
    "street": "123 Main",
    "city": "Lewes",
    "state": "DE",
    "postalCode": "19958"
  },
  "pet": null
}
```

Sample JSON Record

The following JSON object will be used for all examples. This reflects a single entry from a **dns.log** file created by the **Zeek Network Security Monitoring** platform. All examples assume the compact version of this record in a file named dns.log. If you'd like to test on your own, download the sample record from **for572.com/dnslog-sample**.

```
{
  "ts": 1602265824.123071,
  "uid": "CHFRflzsgM15k9et4",
  "id.orig_h": "192.168.75.169",
  "id.orig_p": 58506,
  "id.resp_h": "192.168.75.1",
  "id.resp_p": 53,
  "proto": "udp",
  "trans_id": 50763,
  "rtt": 0.022633075714111328,
  "query": "www.sansgear.com",
  "qclass": 1,
  "qclass_name": "C_INTERNET",
  "qtype": 1,
  "qtype_name": "A",
  "rcode": 0,
  "rcode_name": "NOERROR",
  "AA": false,
  "TC": false,
  "RD": true,
  "RA": true,
  "Z": 0,
  "answers": [
    "vhost1.identityvector.com",
    "70.32.97.206"
  ],
  "TTLs": [
    3600,
    3600
  ],
  "rejected": false
}
```

To learn what these fields mean, see **for572.com/dnslog-fields**

Fundamental Usage: Pretty Print

In its simplest usage, jq will format compact JSON objects into their expanded form, as shown in the panels to the left. There are several fundamental command line options that will help you as well. The `'.'` filter represents the root of each JSON object and will simply display all fields in the object when used.

jq Format, filter, and transform JSON data

```
$ jq [options] <filter> <input_file>
```

-c Compact output instead of "pretty-printed"

-r Raw output instead of quoted JSON text

-s Sort output lexically based on key names

For example, to print the sample **dns.log** entry shown in expanded form on the left:

```
$ jq '.' dns.log
```

Filtering: Just the Field You Want

Many times, the user only needs to display specific fields from each JSON object instead of the entire set. This requires a more detailed filter statement.

To display the value for just one field, identify the field with the `'.<fieldname>'` syntax (note the leading dot).

```
$ jq '.query' dns.log
"www.sansgear.com"
```

To display resulting values in their non-quoted raw form, use the `-r` option to the `jq` command.

```
$ jq -r '.query' dns.log
"www.sansgear.com"
```

When referencing a field name that contains any non-alphanumeric character, double quotation marks must be used. This is common with the dot character, which designates nested JSON objects. However, some JSON logs such as Zeek's use it as a part of the field name which then requires double quoting.

```
$ jq '."id.orig_h"' dns.log
"192.168.75.169"
```

Accessing Array Elements

To access a particular element from an array, familiar `[<element_number>]` notation is used. Remember that array indices are zero-based.

Accessing Nested JSON Objects

Nested objects can be accessed by using the dot separator. While the sample Zeek `dns.log` entry does not contain these, the below example uses the original JSON object from this handout with the shell's pipe operator to show that like many other command-line tools, jq can be used with data on STDIN instead of a file.

```
$ echo '{ "name": "Lance", ⏎
    "age": 42, "active": true, ⏎
    "tags": [ "tag1", "tag2" ], ⏎
    "address": { "street": "123 Main", ⏎
    "city": "Lewes", "state": "DE", ⏎
    "postalCode": "19958"}, ⏎
    "pet": null }' | ⏎
  jq '.address.city'
"Lewes"
```

Complex Filtering: Build New JSON Objects

To select more than one field, the syntax reflects assembling a new JSON object. No leading dot is used in this filter syntax. Note that the `-r` option has no effect.

```
$ jq '{ "id.orig_h", query }' dns.log
{
  "id.orig_h": "192.168.75.169",
  "query": "www.sansgear.com"
}
```

Selecting Records Based on Content

By default, jq will process all JSON objects in the input data set. Using the select operator allows you to limit the records processed based on their values.

```
$ jq 'select(.rcode_name == "NOERROR")'
```

This command will produce all records with a value of "`NOERROR`" in the `rcode_name` field.

The additional operators `contains`, `startswith`, and `endswith` are also useful ways to select records.

```
$ jq 'select(.query | ⏎
   contains("sans"))' dns.log
$ jq 'select(.query | ⏎
  endswith(".com"))' dns.log
```

Note however, that the `select()` operator can be very slow, especially on large data sets. Using a preprocessor such as grep (or `zgrep` for `gzip`-compressed JSON) can provide a dramatic performance improvement.

Chaining `jq` Operations

The pipe symbol, `|`, can be used in the filter statement to pass the output of one operation to the input of the next.

```
$ jq 'select(.rcode_name == "NOERROR") ⏎
  | .query' dns.log
"www.sansgear.com"
```

Reformatting Time Stamps

Many logs use a form of the UNIX Epoch timestamp. Rather than use external conversion utilities, `jq` can convert these natively. (The `|` operator passes output from one part of the filter as input to the next and `|=` replaces a field's value in place.)

```
$ jq '.ts |= todate | .ts' dns.log
"2020-10-09T17:50:24Z"
```