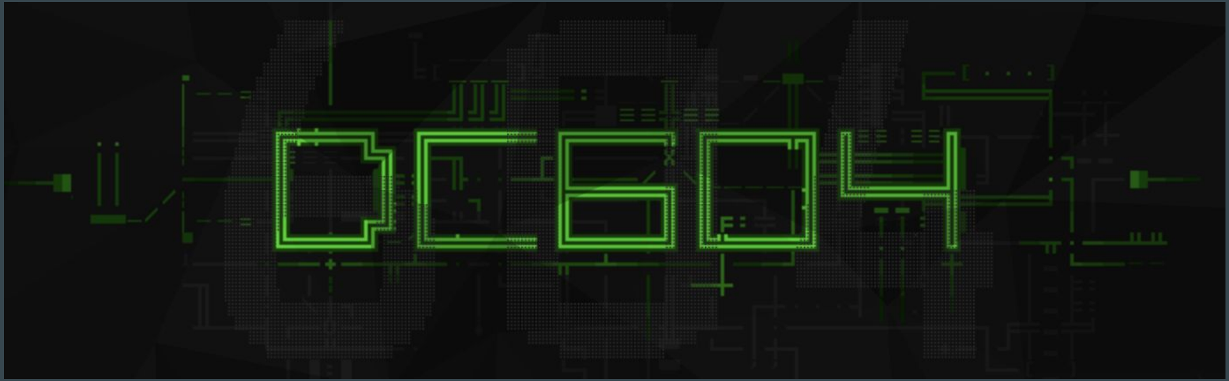
JSON parsing with jq

•••

@airman604 for @Defcon604

DefCon 604 aka DC604

- Local (Vancouver, BC) security group
- Monthly technical talks and workshops, last Thursday of the month
- Join the discussion in #dc604 channel on MARS Slack (https://fourthplanet.ca/slack/)
- Get involved - we're always looking for new content!
- Follow on Twitter @Defcon604
- dc604.ca



JSON

JSON = JavaScript Object Notation.

JSON is a text-based language independent format for storing and exchanging data.

JSON files consist of the following elements (data types):

- Number: `25` or `4.2`
- String: `"this is a string"`
- Boolean: `true` or `false`
- Array: `[1, 2, "3", "four"]`
- Object (or dictionary): `{"key": "value"}`
- `null`

JSON Example

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Why JSON?

Widely used due to simplicity of the format and the fact that it is human-readable.

APIs often use JSON, which can easily be consumed by automation tools and scripts.

Great for logs - self-describing log format, no need for complicated parsers. Con - higher volume as each log entry contains full metadata.

jq

`jq` is like awk for JSON data

`jq` is written in portable C, and it has zero runtime dependencies (single binary).

`jq` can mangle the data format that you have into the one that you want with very little effort, and the program to do so is often shorter and simpler than you'd expect.

Installing jq

Linux (might already be installed)

- `sudo apt install jq`
- `sudo dnf install jq`

Mac OS

- `brew install jq`

Windows

- Chocolatey or binaries here: https://stedolan.github.io/jq/download/

Lab 0

Install `jq` now!

Download the dataset: https://bit.ly/34e5myk (https://www.secrepo.com/honeypot/honeypot.json.zip)

Ask for help in Zoom chat!

jq basics

`jq [options] <jq filter> [file…]`

Pretty print: `cat file.json | jq`

(Same as: `cat file.json | jq .`)

Pretty print with “less”: `cat honeypot.json | jq -C | less -R`

Extract specific fields: `cat honeypot.json | jq .ident`

Fields can be nested: `.foo.bar`

Fields referenced with [ ]: `cat honeypot.json | jq -s '._id["$oid"]'`

Slurp and other options

Single JSON vs multiple JSON documents

`-s / --slurp` - read (“slurp”) all inputs into an array and apply filter to it

`-r` - output raw strings, not JSON texts

Lab 1

List all unique event sources in the dataset.

Lab 1

List all (unique) the event sources in the dataset.

Hints:

- Use jq in “pretty print” mode to explore the structure of the dataset
- Identify a field that specifies the event source
- Use jq to extract just the event source
- Use other command line tools to only list unique event sources

Size of an array

```
jq '.[] | length'
```

Input: `[[1,2], "string", {"a":2}, null]`

Output:

```
2
6
1
0
```

length

The builtin **function** length gets the length of various different types of value:

The length of a `string` is the number of Unicode codepoints it contains (which will be the same as its JSON-encoded length in bytes if it's pure ASCII).

The length of an `array` is the number of elements.

The length of an `object` is the number of key-value pairs.

The length of `null` is zero.

Lab 2

How many events are there in the dataset?

Lab 2

How many events are there in the dataset?

Hints:

- Use `length`
- Remember `-s / --slurp`?

More jq magic

Piping: same idea as shell, use output of the previous step as the input to the next

Use quotes!

Flatten arrays (iterate through all the elements of an array): `.[]`

Array indexing and slicing:

```
    .[N]        .[N:M]      .[-N:]

    .[N:]       .[:M]       .[N: -M]
```

Lab 3

Display one sample payload as raw string.

Lab 3

Display one sample payload as raw string.

Hints:

- Use a combination of “slurp” and array indexing.
- Pick `.payload` field
- Output raw string with `-r`

Grouping

`group_by` creates a nested array grouping objects based on specified field value:

```
jq 'group_by(.foo)'
```

Input: `[{"foo":1, "bar":10}, {"foo":3, "bar":100}, {"foo":1, "bar":1}]`

Output: `[[{"foo":1, "bar":10}, {"foo":1, "bar":1}], [{"foo":3, "bar":100}]]`

Input is an array (remember slurp?)!

Creating arrays with []

```
jq '[.user, .projects[]]'
```

Input:
```
{"user":"stedolan", "projects": ["jq", "wikiflow"]}
```

Output:
```
["stedolan", "jq", "wikiflow"]
```

```
jq '[ .[] | . * 2]'
```

Input:
```
[1, 2, 3]
```

Output:
```
[2, 4, 6]
```

Creating objects with {}

```
jq '{user, title: .titles[]}'
```

Input: `{"user":"stedolan","titles":["JQ Primer", "More JQ"]}`

Output:

```
{"user":"stedolan", "title": "JQ Primer"}
{"user":"stedolan", "title": "More JQ"}
```

```
jq '{(.user): .titles}'
```

Input: `{"user":"stedolan","titles":["JQ Primer", "More JQ"]}`

Output: `{"stedolan": ["JQ Primer", "More JQ"]}`

Lab 4

Count events by data source (channel).

Lab 4

Count events by data source (channel).

Hints:

- You’ll need to group by channel
- You can get the channel name from the first element of each group array
- Use `length` to count events in each group

Comma

```
jq '.foo, .bar'
```

Input: `{"foo": 42, "bar": "something else", "baz": true}`

Output:

```
42
"something else"
```

```
jq '.user, .projects[]'
```

Input: `{"user":"stedolan", "projects": ["jq", "wikiflow"]}`

Output:

```
"stedolan"
"jq"
"wikiflow"
```

Lab 5

Check sample structure of payload for each data source (channel).

Lab 5

Check sample structure of payload for each data source (channel).

Hints:

- Group by channel, as in the previous lab
- Use the first element in each group to get channel name and payload sample
- Raw output will make payload more readable
- Using comma (instead of making an object) will make this possible

map and map_values

Apply provided filter and replace values in the input (array for map, object for map_values).

```
jq 'map(.+1)'
```

Input: `[1,2,3]`

Output: `[2,3,4]`

```
jq 'map_values(.+1)'
```

Input: `{"a": 1, "b": 2, "c": 3}`

Output: `{"a": 2, "b": 3, "c": 4}`

select

Output same as input if condition is true, no output otherwise.

```
jq 'map(select(. >= 2))'
```

Input: `[1,5,3,0,7]`

Output: `[5,3,7]`

```
jq '.[] | select(.id == "second")'
```

Input: `[{"id": "first", "val": 1}, {"id": "second", "val": 2}]`

Output: `{"id": "second", "val": 2}`

Lab 6

Split the dataset into separate files. Only use data sources (channel) amun.events, gastopf.events, snort.alerts.

Lab 6

Split the dataset into separate files. Only use data sources (channel) amun.events, gastopf.events, snort.alerts.

Hints:

- Use select with a condition on channel value
- Output payload in raw format
- Don’t try to do this in one command
- Be aware of the input format (array vs multiple objects)

Lab 7

For amun.events, list all victims for attacker IP `61.153.106.24`

Lab 7

For amun.events, list all victims for attacker IP `61.153.106.24`

Hints:

- None

unique, unique_by(path_exp)

The `unique` function takes as input an array and produces an array of the same elements, in sorted order, with duplicates removed.

The `unique_by(path_exp)` function will keep only one element for each value obtained by applying the argument. Think of it as making an array by taking one element out of every group produced by group.

```
jq 'unique'
```

Input: `[1,2,5,3,5,3,1,3]`

Output: `[1,2,3,5]`

sort and sort_by

Sort an array. sort_by uses a specified field in each object in the array for comparison.

```
jq 'sort'
```

Input: `[8,3,null,6]`
Output: `[null,3,6,8]`

```
jq 'sort_by(.foo)'
```

Input: `[{"foo":4, "bar":10}, {"foo":3, "bar":100}, {"foo":2, "bar":1}]`

Output: `[{"foo":2, "bar":1}, {"foo":3, "bar":100}, {"foo":4, "bar":10}]`

reverse

Reverse the input array.

```
jq 'reverse'
```

Input: `[1,2,3,4]`

Output: `[4,3,2,1]`

Lab 8

For amun.events, display top 10 attackers (by event count).

Lab 8

For amun.events, display top 10 attackers (by event count).

Hints:

- Start with the same idea as in Lab 4 (use group_by and length)
- Use sort and reverse
- Use array slice to only leave top 10

Lab 9

For snort.alerts, determine most attacked network ports.

Lab 9

For snort.alerts, determine most attacked network ports.

Hints:

- Base idea is the same as Lab 8.
- Would it be helpful to know if the port is UDP or TCP?
- You can group_by multiple fields: group_by(.field1,.field2)
- For better display, you can combine port and protocol using +

Lab 10

For snort.alerts, determine alert count by priority.

Lab 11

For glastopf.events, determine number of events per request URL.

Regex in jq

The jq regex filters are defined so that they can be used using one of these patterns:

```
STRING | FILTER( REGEX )
STRING | FILTER( REGEX; FLAGS )
STRING | FILTER( [REGEX] )
STRING | FILTER( [REGEX, FLAGS] )
```

where:

`STRING`, `REGEX` and `FLAGS` are jq strings and subject to jq string interpolation;

`REGEX`, after string interpolation, should be a valid PCRE regex;

`FILTER` is one of `test`, `match`, or `capture`.

test(val), test(regex; flags)

Test whether input string matches the regex, return true or false.

```
jq 'test("foo")'
```

Input: `"foo"`

Output: `true`

```
jq '.[] | test("a b c # spaces are ignored"; "ix")'
```

Input: `["xabcd", "ABC"]`

Output:

```
true
true
```

Lab 12

For glastopf.events, list all attempts to exploit shellshock.

Lab 12

For glastopf.events, list all attempts to exploit shellshock.

Hints:

- Full request contained in request_raw
- Test for `{ :;}`

match(val), match(regex; flags)

match outputs an object for each match it finds. Matches have the following fields:

`offset` - offset in UTF-8 codepoints from the beginning of the input

`length` - length in UTF-8 codepoints of the match

`string` - the string that it matched

`captures` - an array of objects representing capturing groups.

Capturing group objects have the following fields:

`offset` - offset in UTF-8 codepoints from the beginning of the input

`length` - length in UTF-8 codepoints of this capturing group

`string` - the string that was captured

`name` - the name of the capturing group (or null if it was unnamed)

Capturing groups that did not match anything return an offset of -1

capture(val), capture(regex; flags)

Collects the named captures in a JSON object, with the name of each capture as the key, and the matched string as the corresponding value.

```
jq 'capture("(?<a>[a-z]+)-(?<n>[0-9]+)")'
```

Input: `"xyzzy-14"`

Output `{ "a": "xyzzy", "n": "14" }`

Regex flags

FLAGS is a string consisting of one of more of the supported flags:

`g` - Global search (find all matches, not just the first)

`i` - Case insensitive search

`m` - Multi line mode ('.' will match newlines)

`n` - Ignore empty matches

`p` - Both s and m modes are enabled

`s` - Single line mode ('^' -> '\A', '$' -> '\Z')

`l` - Find longest possible matches

`x` - Extended regex format (ignore whitespace and comments)

Lab 13

For glastopf.events containing shellshock attacks, extract possible malware URLs.

Lab 13

For glastopf.events containing shellshock attacks, extract possible malware URLs.

Hints:

- Look for URL patterns in `request_raw`
- Regex for URL: `(?<url>https?://[0-9a-zA-Z-._~:@?#+/%]+)`

+

The operator + takes two filters, applies them both to the same input, and adds the results together. What "adding" means depends on the types involved:

`Numbers` are added by normal arithmetic.

`Arrays` are added by being concatenated into a larger array.

`Strings` are added by being joined into a larger string.

`Objects` are added by merging, that is, inserting all the key-value pairs from both objects into a single combined object. If both objects contain a value for the same key, the object on the right of the + wins. (For recursive merge use the * operator.)

`null` can be added to any value, and returns the other value unchanged.

+

```
jq '.a + 1'
```

Input: `{"a": 7}`

Output: `8`

```
jq '.a + .b'
```

Input: `{"a": [1,2], "b": [3,4]}`

Output: `[1,2,3,4]`

```
jq '. + {b: 2, c: 3, a: 42}'
```

Input: `{"a": 1}`

Output `{"a": 42, "b": 2, "c": 3}`

Lab 14

For glastopf.events, determine event count by User-Agent.

Lab 14

For glastopf.events, determine event count by User-Agent.

Hints:

- User-Agent can be extracted from request_raw
- Use regex and ignore case: `.*user-agent: *(?<userAgent>.*)(?:\r|$)`
- Use `+` to add a new `userAgent` field to each event object
- The rest is similar to previous labs

THANK YOU!