DJ - Dictionary Juggler

A domain-specific language (DSL) for analyzing, transforming and generating dictionaries for password recovery.

Dr. Michael Eichberg

Summer 2023

Motivation

When dealing with dictionaries...

1. we want to do similar things very often:
    i. generate (parts of) new dictionaries and simultaneously corresponding (Hashcat) rules
    ii. transform existing dictionaries to fit a specific case or device policy
    iii. filter existing dictionaries due to technical reasons (e.g. only 7bit ascii is supported, the password requires passwords with at least...)
    iv. analyze leaks to get a better understanding of the current trends

Motivation

1. doing things using a combination of `tr`, `sed`, `awk`,`grep`, *shell scripts*, *python hacking*, `hashcat --stdout`, ... is possible (at least to some extent), but neither reusable nor comprehensible (at least not after some months, weeks, days, ... *intermediate files "hell"!
2. **We need/want**:
    1. **comprehensible documentation** what was done
    2. **repeatability** of the build process
    3. to **generate various artifacts**

Example - Processing a "real" dictionary

Given the entry "Audi RS", you want to generate:

```
Audi-RS
Audi_RS
audi-rs
audi_rs
AudiRS
audirs
Audi
audi
RS
rs
```

DJ:

```
+split " " +remove_ws *map " " "-_" +lower report
```

Example - Comprehending a Leak

Identifying the structure of passwords (for subsequent statistical analysis)

- password => llllllll
- Password => ulllllll
- Password2023! => ulllllllddddds

DJ:

```
*map "abcdefghijkmnopqrstuvwxyz" "l" \
*map "ABCDEFGHIJKLMNOPQRSTUVWXYZ" "u" \
*map "0123456789" "d" \
*map "^°!\"§$%&/()=?`{[]}\´+*~#'-_.:,;µ@€|<>" "s" \
*map not "luds" "?" \
report
```

Example - Simple Rule Extraction

A (poor man's ?) rule extractor:

- Password2023! => `$2$0$2$3$!`

  DJ: `find_all "[^a-zA-Z]+$" prepend each "$" report`
- 1Password => `^1`

  DJ: `find_all "^[^a-zA-Z]+" *reverse prepend each "^" report`

Example - Generating Alternatives

Given some simple "specification", e.g.: `[["F"],["3","è",""],["st","St"]]` we want to generate the alternatives:

```
F3st
Fèst
Fst
FSt
...
```

DJ:

```
gen alt [["F","f"],["E","e","3","é","è",""],["st","St","sT"]]

max upper 1 report
```

Example - finding German words

Given a leak we want to identify the German words that are (frequently) used:

DJ:

```
#[optional] ignore "ignore/de.txt"
config is_regular_word DICTIONARIES ["de"]

is_regular_word report
```

This covers only the most basic scenarios, but it gives a first impression.

Example - Filtering Leaks

Filtering passwords that are most likely not reused and which should not be in a **focused** dictionary; corresponding examples in Rockyou:

- a3eilm2s2y
- K3F3H6I9
- 0p9o8i7u
- c9p5au8na
- !@#$%^&
- ...

Conceptual Idea

1. Process, i.e., transform, filter, take apart or analyze, ***one entry of a dictionary at a time*** to create the needed entries.
   This enables DJ to process arbitrary large dictionaries.
2. Processing an entry is done by simple, basic operations that can be chained.
3. Enable the processing of every entry using multiple pipelines to create - based on the same entry - all desired variants.
4. Apply (basic) operations (e.g., split, strip whitespace, lower, ...) on *all results* of the previous operation.
   I.e., an operation may produce between zero and many (intermediate) results.
5. Write out the results (to a file or stdout).

Some selected features require memory which scales along with the size of the dictionary.

Basic example (`split " " report`)

Operation	Input	Output	Remark
`split " "`	["A Test"]	["A","Test"]	
`report`	["A","Test"]	["A","Test"]	Additionally, printed to stdout

Running it on the shell:

```
/DJ% echo 'A Test\nAnother Test' | ./dj.py 'split " " report'
A
Test
Another
Test
```

Basic Grammar

```
op_seq := (<operation_name> <operation parameters>?)+

operation_name := "[a-z_]+"
operations parameters := ...defined by the operation,
                            but the last parameter must
                            never match another operation's name!
```

Output

- Either a `report`, `classify` or `write` operation is required to output the results of an operation sequence.
- Multiple write operations to the same file are possible.
- To create pre-initialized files use the `create` directive.
- To suppress duplicates use the parameter `-u`.
  This requires that the generated dictionary completely fits into memory!

Basic Usage

```
dj.py [-h] [-o OPERATIONS] [-d DICTIONARY] [-v] [-t] [--progress] [--pace] [-u] [OPs ...]
```

- *Command-line usage*: the operations are directly specified and the dictionary is read from the command line:

  `echo 'A Test\nAnother Test' | ./dj.py 'split " " report'`

- *Scripted*: The operations script and (optionally) the dictionary are read from a file:

  `./dj.py -o Scripts.dj -d Towns_of_the_world.txt`

Operations that "*do not apply*" to an entry will filter it!

Apply generally means, the operation has some effect on the given entry.

Operation	Input	Output	Remark
`get_no`	["acap"]	None(N/A)	Terminates the processing pipeline.
`lower`	["amsterdam"]	None(N/A)	Terminates the processing pipeline.

Details about "applicability" of an operation is found in the documentation. In general: a transformation that makes no changes or an extractor that does not extract anything are considered as being not applicable; filter operations are always applicable.

Applicable Transformers

When applying an operation, DJ distinguishes two cases: An *operation is applicable* vs. *an operation is not applicable*; this facilitates advanced processing of entries. In particular, it enables us to continue with the processed entry if the operation was applicable and otherwise continue with the original entry.

Operation	Input	Output	Remark
`remove "-"`	["---"]	[""]	The empty string will be then ignored

Operation	Input	Output	Remark
`remove "-"`	["AB"]	(N/A)	None will then be ignored.

All filters are always considered being applicable.

Modifying the behavior of a not applicable op: `+` Meta-Operator

- `+` (e.g., `+remove "-"`) will ensure that the result of the operation will always contain the original entry and the results of the transformation.
- useable for Transformers and Extractors

Modifying the behavior of a not applicable op: `*` Meta-Operator

(*Continue if not applicable*)

- `*` (e.g,. `*remove " "`) will pass on the original entry, if and only if the operation was not applicable to a given entry.

Operation	Input	Output	Remark
`*remove "-"`	["a-b"]	["ab"]	
`*remove "-"`	["ab"]	["ab"]	The operation was not applicable.
`*remove "-"`	["---"]	[""]	The operation was applicable.

Modifying filters: `!` Meta-Operator

- `!` (e.g., `!is_pattern`) inverts a filter operation.
- useable only by filters

Some Basic Transformers

Shorten	Extend	In-place	Split-up	Special
remove	append	number	split	correct_spelling
fold_ws	prepend	replace	sub_split	related
deduplicate	multiply	title	segments	mangle_dates
deduplicate_reversed		upper	break_up	_ ("NOP")
detriplicate		rotate	find_all	
strip		capitalize	get_no	
…		…	…	

Filter Operations

`is_part_of`, `is_pattern`, `is_walk`

`is_popular_word`, `is_regular_word`

`min`, `max`, `has`

`sieve`

Logical Operators

`or`

Lists

- The implicit (current) per dictionary entry list (`ilist\_*`)
- Named global lists, which contain pre-loaded data used by selected operations
  (Early development phase!)
- Named per dictionary lists enable a step-wise processing.

Named lists

- Named lists first need to be declared and then need to be populated by operations.
- Named lists can be populated in a piece-wise manner.
- Named lists are used using the built-in `use` operation.
- Named lists are cleared after processing a dictionary entry.

Using a named list

In the following example, we declare and use a list `DATES` to store extracted dates.

DJ

```
mangle_dates \
    { write "dates.txt" }> DATES

use DATES \
    prepend each "$"                write "append_dates.rule"

use DATES \
    reverse prepend each "^"        write "prepend_dates.rule"
```

Intermediate List

Operations beginning with "ilist_" operate on the current, implicit, intermediate list:

- **Tests**: `ilist_if_else`, `ilist_max`, `ilist_ratio`
- **Quantification over the intermediate list**: `ilist_if_all`, `ilist_if_any`
- **Transformations**: `ilist_concat`, `ilist_unique`
- **Passing all current entries to the next operation one ofter another**: `ilist_foreach`

Using intermediate and named lists

Example to create rules that prepend and append at the same time (e.g., which create the rule `^1 $$` from the password "1password$")

```
list LEFT
list RIGHT
{ find_all "^[^a-zA-Z]+" *reverse prepend each "^" }> LEFT
{ find_all "[^a-zA-Z]+$" prepend each "$" }> RIGHT
use LEFT RIGHT ilist_max length 1 report
use LEFT RIGHT ilist_concat " " report
```

(Global) Configuration

- Many operations have per instance configuration options, but some also have global configuration options. In general, instance configuration options are always used if it is imaginable that the same operation is used multiple times in a script with different options.
- Global configuration options are used for those options that are not related to processing individual entries. (E.g., setting the languages that we are interested in, configuring the "relatedness", ...)

Further Topics

- looping (already available and stable, but beyond this initial presentation)
- consolidation of operations (currently we have 4 strip operations...)
- a big cheat-sheet (I'm working on it...)
- documentation how to extend DJ
  (For now, just call me and you'll have your feature in a ... or feel free to extend it; I'm happy to take pull requests?)
- versioning
  (As soon as we have stabilized the feature set, we will have versioning for the scripts... for now track the changes in GIT to understand what needs to be done to old scripts to get them working again... sorry!)