CPE Ontology Generator

Vladimir Dimitrov [0000-0002-7441-253X]

Sofia University „St. Kliment Ohridski", Faculty of Mathematics and Informatics
1164 Sofia, 1 James Bourchier Blvd., Bulgaria

cht@fmi.uni-sofia.bg

Abstract. This product has been developed as an automatic tool for generation of CPE ontologies from NIST Official Common Platform Enumeration Dictionary. It is written Python and is available at GitHub.
The tool downloads dictionary from its site. It is in XML format – compressed as Zip file. Then decompress it and generate a CPE ontology.
NIST CPE Dictionary is very huge file that permanently grows. Its processing require all server resources to be used (computing power and memory). The tool is using in concurrent (parallel) mode all available CPUs and their cores available on a single server. Currently, there is no need to expand the execution on several severs but the execution model can be easy extended to such a variant.
For every available processor core, a worker process is generated. The main process parses the dictionary and put every CPE item description in a common worker input queue. The worker process consume a CPE parsed XML description and generate an OWL individual for it. Generated individuals are queued on a common worker output queue that is consumed by a "writer" process that write it contents is a single file in OWL Manchester Syntax. Only one ontology file is generated.
Synchronization among processes is achieved via the queues. A process is blocked when it tries to queue read in or write out, and the queue is full. This is pipeline with several parallel workers, one source and one sink.
Finally. The main process signal via the queue the end of processing and wait for all other processes to finish their execution.

Keywords: CPE, OWL, Ontology, Cybersecurity, Python, Generator.

1 Introduction

The CPE dictionary of platform names is freely available on the NIST website [1]. It is designed to serve as a naming base for other cybersecurity databases, such as the databases for vulnerabilities, vulnerabilities and patterns of attacks.

The initial idea for vulnerability description in CVE [2] has been to follow a semi-structured pattern identifying the main vulnerability characteristics. This recommendation is still valid, but recognizing the individual characteristics in the phrase is a problem even for a cybersecurity specialist.

An attempt for automatic recognition of these phrases has been made in [3].

This vulnerability description pattern may include vulnerability type (CWE [4] according to the MITRE Corporation classification, if possible), platform component, supplier, product, version, root cause, attacker, influence and attack vector.

The main problem in the recognition process is the retrieval of the supplier and product names. Unfortunately, the available databases that can help to do that do not contain such standardized information. For example, IBM or International Business Machine Corporation? A typical example of such a database is [5]. The quality of the data in this database is extremely poor – manufacturers' data are mixed with those for products, and any spelling of names is included.

Another fundamental problem is the fact that this description pattern of CVE is not followed. The reasons for this are various – from the lack of information to the inability to fit into this pattern, but this is the actual situation.

In that situation, it was necessary to create a unified classification of CPE platforms [6], according to which CVE vulnerabilities can be classified. CPE is a development of MITRE Corporation, but is currently maintained by NIST. Vulnerabilities in NVD [7] are classified by CPE [1], while those in MITRE Corporation are not. The vulnerability database of MITRE Corporation is community-based. Cybersecurity community adds and examines newly registered vulnerabilities, for some of them the information is partial. The NIST vulnerability database contains only those vulnerabilities for which sufficient information has been collected, and they have been further processed and enriched by NIST. In particular, the additional processing classifies vulnerabilities by platforms using CPE names.

CPE is a publicly available dictionary of platform names. For each platform, it contains information about a part (rather a type of platform: hardware, operating system or application), vendor, product, version, update, edit, language (of the interface), software edition (for a specific environment), target software (on which it runs), target hardware (on which it runs), and other. The values of these attributes of CPE names are standardized according to the specification [8].

The CPE Names Dictionary is freely available for use from NIST site in XML format. This dictionary can be extended by the users in accordance with their needs, but the recommendations are when doing so to follow the CPE specifications.

In fact, the CPE specification is the building block on which the other databases of vulnerabilities, vulnerabilities and attack patterns of NIST are built.

Based on the CPE specifications, an ontology has been developed that allows CPE names to be used in an OWL environment. A publication on CPE ontology is forthcoming.

The subject of this paper is the OWL ontology generator for CPE names.

2 The implementation

The CPE ontology generator is implemented in Python 3.9. It is published in GitHub at [9]. It is in the form of a Python script `generateCPEontology.py` and a shell (set of axioms) of the ontology in a `shell.owl` file. The shell is a description of the ontology that, in our case, is modified by the generator and filled with individuals. It can be opened in Protégé for viewing and editing.

The generator, shell, and work files are in the same directory. The generation creates the ontology in a file named `cpe23.owl` in the same directory.

Before generating the ontology, the generator downloads the dictionary from the NIST website (`official-cpe-dictionary_v2.3.xml.zip`) and unzips it to the `official-cpe-dictionary_v2.3.xml` file.

To avoid frequent visits to the NIST site, the dictionary file is not downloaded by default. If there is no downloaded and unzipped copy, the script will give an exceptional error for the lack of an input file.

To cause the dictionary file to be downloaded from the NIST site, the `-d` or `-download` parameter must be passed to the script. The `argparse` package is used to process the parameters. This is done globally in the module.

The CPE vocabulary is huge. On March 2021, it contains over 600 000 names and is growing steadily.

The dictionary is loaded from the site by the function `downloadCPE23()`. It also performs above mentioned unzip. No problem if there are old copies of the files – they are just deleted. This function uses the `urllib.request` and `zipfile` packages.

The description of the CPE dictionary is made in the following three schemes: CPE 2.3 XML Schema (`cpe-dictionary_2.3.xsd`), CPE 2.3 Dictionary Extension XML Schema (`cpe-dictionary-extension_2.3.xsd`) and CPE 2.2 XML Schema (`cpe-dictionary_2.2.xsd`).

Additionally, there is a description of the NIST CPE Metadata 0.2 XML Schema metadata (`cpe-dictionary-metadata_0.2.xsd`).

These XML schemas are available from [1].

The concept of the CPE version 2.3 design is to be an extension of the schema of version 2.2 through the element `xsd:any`. In fact, the CPE 2.3 XML Schema follows CPE version 2.2, and its extensions appears in the second CPE 2.3 Dictionary Extension XML Schema.

The third CPE 2.2 XML Schema is just the old version of the schema.

The effect of this solution is that an application that is working with CPE version 2.2 would continue to work with the dictionary of the newer version, practically ignoring its extensions.

A detailed description of the dictionary and XML schemas can be found in the specification [11].

After loading the dictionary and unzipping it, the generator loads the contents of the dictionary in the main memory and parses it. This is done with the function `parseXML()` from the package `xml.etree.ElementTree`.

The next step is the initialization of the ontology file (`shell.owl`) with annotations using some of the information fields. This is done by the function `generateShell()`.

After generating the ontology description, the actual generation of its individuals begins. This is the task of the function `generateIndividuals()`.

A critical problem in the individual generation is the dictionary volume. In sequential processing, only one processor core is used and the operation lasts several hours. For example, on the development system it lasts about 8-9 hours. All this time, the other processors and cores are idle or under minimal load.

The above considerations necessitated the conversion of the generator into a parallel version, which uses all the processors and cores. For this purpose, the package `multiprocessing` has been used, which realizes real parallelism in Python. In fact, each process with this package is hooked up to a separate core.

The development has been done on Windows 10, and it is not tested in another environment. However, in accordance with the package (`multiprocessing`) description, it should work in parallel on every Python port without any problems.

The function `generateIndividuals()` creates two `inQueue` and `outQueue` queues and a process that writes individuals to the ontology file. The last process reads individual descriptions from the queue `outQueue` until it receives an end signal. The queue transmits individual descriptions as character strings (serialized). Then the writer process adds these descriptions to the ontology file. When a character string with a value of "DONE" is read, the process closes the ontology file and completes its execution. The code of this process is in the function `writeResults()`. It opens the ontology file in the add-in mode, because the function `generateShell()` closes it when finishes its work.

The function `generateIndividuals()` creates as many parallel processing processes as there are cores available on the computer. The code of these processes is represented in the function `generateIndividual()`.

Each of these processes reads from the queue `inQueue` a description of CPE name in a serialized, parsed XML format and generates in the output queue `outQueue` the description of the received individual.

In the main process, `generateIndividuals()` fills the `inQueue` queue with serialized XML descriptions of CPE names. If the queue becomes full, the process blocks until a space is released in the queue. If the queue `outQueue` is full, then the processes `generateIndividual()` block until the process `writeResults()` releases a space in the queue.

The processes `generateIndividual()` have a built-in synchronization mechanism when reading from the input queue `inQueue`, as well as when writing to the output queue `outQueue`. The same goes for the processes `generateIndividuals()` when writing in `inQueue` and for `writeResults()` when reading from `outQueue`.

In parallel processes `generateIndividual()` there is another feature related to parallelism. This is the `CPE23Names` parameter, which is a shared dictionary of type `Manager`, and it contains for each of the dictionary entries a key with FS binding the CPE name with a value its URI binding. The dictionary is prepared by the function `getCPE23Names()` and passed at process initialization. Dictionary reading (`Manager`) has a built-in synchronization mechanism. The need for this dictionary is dictated by the fact that the FS name is the priority name in the CPE dictionary, while the URI name is rather a parameter in the CPE name description, while in CPE ontology the URI name is used as individual identifier.

The scheme of the parallel processes is shown in Fig. 1.

Each of the processes `generateIndividual()` analyzes the XML description of the CPE name and creates an individual description for the OWL ontology. Part of the data goes into annotations, and another part – in data properties.

It is more complicated with obsolete CPE names, where not only data properties are generated, but an object property pointing to individual of class `Deprecation` is created. The last one describes the facts about the name replacement or removal. A major problem in generating `Deprecation` individuals is the creation of the `deprecated-by` object properties, since the replacement of CPE names are not with "basic" CPEs, but search patterns are used according to [12]. These patterns are extended to sets of "basic" CPE URI names with the function `getByWildCards()`. The latter works with the `CPE23Names` dictionary and uses regular expressions based on the CPE Match pattern. The FS names (keys) that correspond to the regular expression are selected from the dictionary, and the URI names (their values) are loaded into the resulting set.

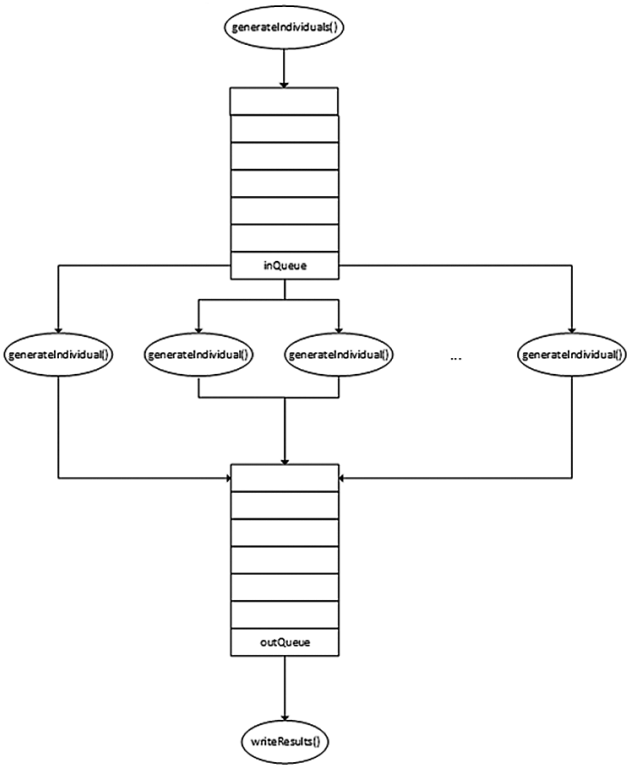

Fig. 1. Parallelism schema.

Some patterns do not match any basic CPE names. What is the reason for this is not clear, but it is not rare case.

When the process `generateIndividuals()` has finished loading in `inQueue` XML descriptions of CPE names, it write there one "DONE" message for each of `generateIndividual()` processes. The latter ones loops from until the character string "DONE" is read from the queue. In Python, it is very convenient – there is no strict type checking and any objects (serialized) can be passed in the queues. In the case of `inQueue`, both serialized XML objects and the regular "DONE" string are passed. Type and value checking is built-in in Python.

After sending the "DONE" signals to the `generateIndividual()` processes, the main `generateIndividuals()` process waits (synchronizes) with them to complete – they continue to run until the queue `inQueue` is exhausted.

Finally, `generateIndividuals()` sends a "DONE" message to the queue `outQueue` to inform the process `writeResults` that the job is com-

plete and waits (synchronizes) for the process to complete. The latter ends after the queue `outQueue` is exhausted. This completes the operation of the generator.

3 Conclusion

The process `generateIndividual()` can be initialized on other computers accessible over the network. Dictionary (class `Manager`) and `Queue` sharing mechanisms are also applicable in the case of distributed processing without modification. At least that is according to the documentation.

Of course, it will be necessary to specify the URLs of the respective machines, which is not done in the current version.

Such an extension has not been made, as for now the generator successfully copes within reasonable limits with the generation of the order of 1-2 hours on a desktop computer with a quad-core processor.

4 Acknowledgements

I would also like to thank NIST for consulting on certain issues in the dictionary and especially to Amy Mahn for her responsiveness.

This work was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

This research is supported by the National Scientific Program "Information and Communication Technologies for a Single Digital Market in Science, Education and Security (ICTinSES)", financed by the Ministry of Education and Science.

References

1. NIST, NVD (National Vulnerability Database), Official Common Platform Enumeration (CPE) Dictionary, https://nvd.nist.gov/products/cpe, last accessed 24/04/2021.
2. MITRE Corporation, CVE, https://cve.mitre.org, last accessed 24/04/2021.
3. Dimitrov, V., Chapter Two. CVE Annotation, in Information Security in Education and Practice, editor K. Kaloyanova, Cambridge Scholars Publishing, 2021, ISBN (10): 1-5257-6066-X, ISBN (13): 978-5375-6066-6.
4. MITRE Corporation, CWE (Common Weakness Enumeration), https://cwe.mitre.org
5. Özkan S., CVE Details, https://www.cvedetails.com, last accessed 24/04/2021.
6. MITRE Corporation, CPE, https://cpe.mitre.org, last accessed 24/04/2021.
7. NIST, NVD (National Vulnerability Database), https://nvd.nist.gov, last accessed 24/04/2021.
8. NIST, NISTIR 7695, Common Platform Enumeration: Naming Specification Version 2.3, https://csrc.nist.gov/publications/detail/nistir/7695/final, last accessed 24/04/2021.
9. Dimitrov V., CPE-ontology-generator, https://github.com/VladimirDimitrov1957/CPE-ontology-generator, last accessed 24/04/2021.

10. Musen, M.A. The Protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015. DOI: 10.1145/2557001.25757003.
11. NIST, NISTIR 7697, Common Platform Enumeration: Dictionary Specification Version 2.3, https://csrc.nist.gov/publications/detail/nistir/7697/final, last accessed 24/04/2021.
12. NIST, NISTIR 7696, Common Platform Enumeration: Name Matching Specification Version 2.3, https://csrc.nist.gov/publications/detail/nistir/7696/final, last accessed 24/04/2021.