ГОДИШНИК НА СОФИЙСКИЯ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Том 111

ANNUAL OF SOFIA UNIVERSITY “ST. KLIMENT OHRIDSKI”
FACULTY OF MATHEMATICS AND INFORMATICS
Volume 111

---

# CAPEC ONTOLOGY GENERATOR

Vladimir Dimitrov

CAPEC is an effort coordinated by MITRE Corporation. Its aim is attack pattern database structured in taxonomies. CAPEC is available as XML document from its project site. CAPEC structure and content are under permanent change and development. It is still not mature database but may be never will.
CAPEC, CWE, and CVE are databases devoted to attacks, weaknesses, and vulnerabilities. They refer each other forming a knowledge ecosystem in cybersecurity area.
Traditional approach for knowledge presentation as information does not work well with conceptualizations under dynamics of this ecosystem and particularly of CAPEC. In this paper, an alternative approach to CAPEC knowledge presentation is proposed, as ontology. First, CAPEC structure and content are discussed and then ontology structure is introduced. CAPEC as ontology opens doors to “open world” concept that is more adequate to ecosystem dynamics.
CAPEC ontology is programmatically generated from CAPEC database.
CAPEC ontology generator in implemented in Python.

**Keywords:** cybersecurity, attack patterns, ontology, CAPEC, OWL, Python

**2020 Mathematics Subject Classification:** 68M25, 68T30, 68U35

**CCS Concepts:**

- Security and privacy∼Formal methods and theory of security∼Formal security models;
- Security and privacy∼Formal methods and theory of security∼Logic and verification

## 1. Attack patterns

CAPEC (Common Pattern Enumeration and Classification) [5] is an effort coordinated by MITRE Corporation. Its aim is attack pattern database structured in taxonomies.

CAPEC is freely available in XML format from the site.

MITRE Corporation maintains two more initiatives CVE [4] and CWE [11]. These are vulnerability and weakness databases.

DOI: 10.60063/GSU.FMI.111.25-33

CVE is mature enough to be maintained officially by NIST as NVD [7].

Original CVE is still maintained by MITRE Corporation for new vulnerability registration.

NVD contains only analyzed vulnerabilities augmented with additional analytic information supplied by NIST.

CWE and CAPEC are still not mature. They are under active development.

CVE, CWE, and CAPEC refer to each other forming knowledge ecosystem in cybersecurity area.

Vulnerabilities are revealed errors, faults, or gaps in specific products fixed by its version and execution environment. CVE entries are numerous – currently 197676 in March 2023.

Weaknesses (CWE) are vulnerability types. Vulnerability processing includes type assignment – one or more CWEs are associated with vulnerability under consideration.

Attacks (CAPEC) exploit vulnerabilities. More precisely, attack patterns exploit one or more weaknesses.

CWE and CAPEC have structures organized by several taxonomies, while CVE is simply a list.

CWE and CAPEC are still not mature and they contain some discrepancies in their notation, structure and content.

CWE and CAPEC are distributed as XML documents, while CVE – as JSON documents.

Our approach is to present CVE, CWE, and CAPEC in OWL [12]. Semantic web is more suitable for formal knowledge presentation than XML. This is especially valid when the knowledge base is under development – not mature.

The term “formal knowledge presentation” refers to conceptualized knowledge presentation, for example via OWL.

CVE, CWE, and CAPEC ontologies are presented in several other publications.

Subject of this paper is CAPEC ontology generation from its XML presentation.

## 2. CAPEC ontology generator

CAPEC ontology generator does not require parallelism. On 02.01.2023 attack patterns are 555.

CAPEC ontology generator is published at the following address `https://github.com/VladimirDimitrov1957/CAPEC-ontology-generator`. It is implemented in Python.

CAPEC generator presentation below follows its control flow.

The generator by default works with input data downloaded in “`data`” subdirectory. With parameter “`-d`” or “`-download`” fresh copy of CAPEC database can be downloaded from CAPEC site.

Procedure `main(download)` manages the whole generation process.

Procedure `downloadCAPEC()` downloads database copy from CAPEC site. This copy is compressed in zip format, so the procedure decompresses downloaded file. CAPEC database is XML document.

Every time validation of input XML document is done using CAPEC XSD schema. For this purpose, `lxml` package is used. It is important to match used packages because some of them simply does not work or are not usable.

In the next step, XML document presenting the database is parsed to internal format. This parsed CAPEC database is used in the following operations.

Procedure `generateIndividuals(root)` performs CAPEC ontology generation. Parameter "`root`" presents CAPEC database in parsed internal format.

This procedure copies and modifies ontology shell from file `shell.owl`. The result is written in file `capec.owl`. External references in the last file are presented as annotations.

Then procedure `generateIndividuals(root)` generates attack patterns, categories, and views with:
`generateAttackPatternIndividual(item, out_file)`,
`generateCategoryIndividual(item, out_file)`, and
`generateViewIndividual(item, root, out_file)`
presented below.

All catalog elements are scanned and corresponding generation procedure is called.

Finally, procedure ends with generation of object property objects. It is described in more details in subsections about `AttackPattern`, and `Individual` class. Figure 1 shows class structures.

Procedure `generateAttackPatternIndividual(item, out_file)` has parameter "`item`" that contains an internal XML presentation of an attack pattern. For this attack pattern, the procedure generates an individual description.

Parameter "`out_file`" is the file in which the ontology is written. Here, more specifically, the attack pattern individual description is written.

`AttackPattern` class plays central role for attack pattern individual generation. An object of this class collects all individual characteristics. Then these characteristics are serialized as a string and written in the ontology file. The class is described in separate section below.

Procedure `generateCategoryIndividual(item, out_file)` is simple. It follows category subelements structure and generates a category individual description.

Procedure `generateViewIndividual(item, root, out_file)` is hardcoded by catalog views. The last are very specific to be implemented in some common generation algorithm.

`Individual` is the other class in CAPEC generator. It collects information about the target objects for object properties. This class has extent, i.e. supports set of all `Individual` objects.

At the end of procedure `generateIndividuals(root)`, objects from `Individual` extent are serialized and appended to the result file.

Class `AttackPattern` and `Individual` are presented in Figure 1 as UML class diagram.

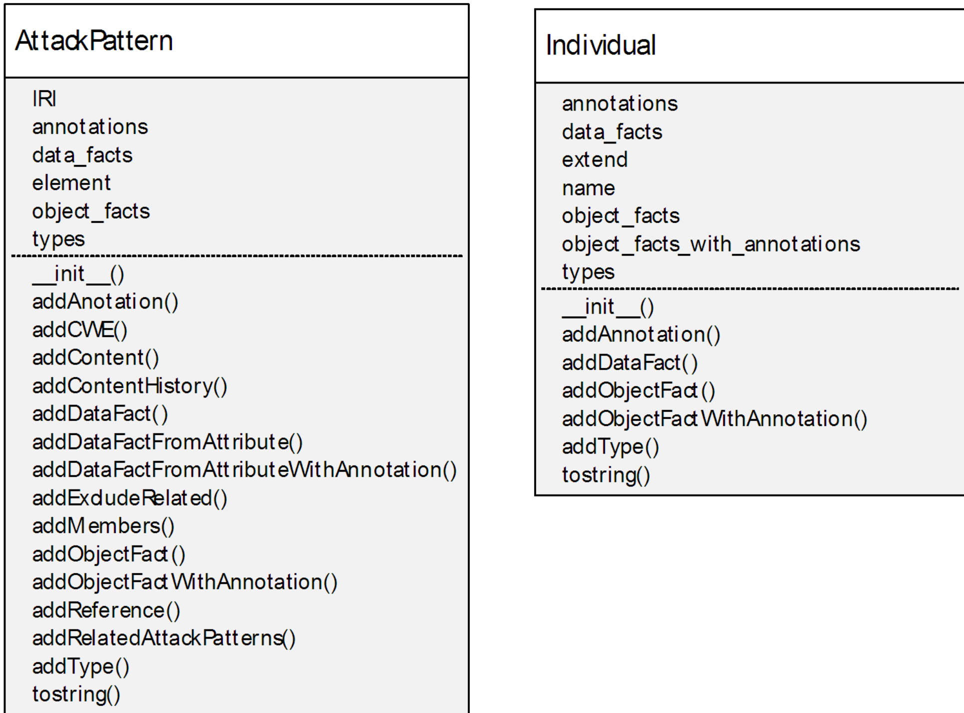


Figure 1. CAPEC generator classes

### 2.1. AttackPattern class

`AttackPattern` instance variables are "`element`" – internal XML presentation of attack pattern, individual IRI, dictionary of annotations, dictionary of data facts, dictionary of object facts, and set of individual types. Data facts present individual data property instances. Object facts present individual object property instances.

When an object of the class is instantiated, its instance variables are accordingly initiated.

Method `addType(self, aName)` adds a type to the individual type set. This operation is hardcoded for categories because their type is more specifically extracted.

Method `addDataFact(self, tag, path = "", structured = False)` adds a data fact.

The data fact is extracted from a subelement of "`element`". The relative path to this subelement is given by parameter "`path`".

Content of a data fact element can be structured text. It is marked by the parameter "`structured`".

If data fact content is structured text, function `stext(s, tag)` is used to remove line formatting and codes the text as OWL literal. For the last operation,

function `flat(s)` is used, but that function is used regularly for non-structured text.

Method `addDataFactFromAttribute(self, att)` extracts data fact from "`element`" attribute. Parameter "`att`" contains attribute name and it is a data property name too.

Method `addDataFactFromAttributeWithAnnotation(self, el, att, path, aName)` extracts data fact from attributes of "`element`" subelements. Parameter "`el`" contains a subelement tag. If the subelement is not a child of "`element`", then the relative path to it is given by parameter "`path`".

It is possible the subelement to have several instances. In that case, the data fact should be with several values.

Parameter "`aName`" is a dictionary with keys that are data fact values. Dictionary value is a list of annotations applicable for this data fact. In this method, only list with one annotation name is used, but presentation is unified with all other methods.

Method `addDataFactWithAnnotation(self, tag, aTag, path = "", name = None, aName = None, structured = False)` is as previous one, but it extracts data facts from "element" subelements.

Attributes are always extracted from subelement content – not from attribute values.

Parameters "`tag`" and "`aTag`" contents are data fact and annotation element tags correspondingly. In most of the cases, tag and data property share the same name. The same is valid for annotation element tag and annotation name. If it is not true, then data fact and annotation name are set by parameters "`name`" and "`aName`".

Parameter "`path`" is used when subelements are not children of "`element`". This is as in the previous method.

Parameter "`structured`" has the same purpose as in method `addDataFact (self, tag, path = "", structured = False)`.

Method `addAnnotation(self, tag, name = None, path = "", structured = False)` extracts annotations form "`element`" subelements. If the annotation name differs from the subelement tag, then parameter "`name`" is used. The other parameters have the same meaning and usage as in above methods.

Method `addReferences(self)` is more specific. It extracts references from "`element`" – from subelements `References` and `Reference`. Annotations are formatted as literals. All references are annotations.

Method `addContentHystory(self)` is specific. It extracts and adds as annotations the content history.

Method `addObjectFact(self, path, oName, cName, cADict)` extracts object facts form "`element`" subelements.

If subelements are not "`element`" children, then parameter "`path`" sets the relative path to them.

Parameter "`oName`" contains the object property name.

The target individual type is given by the parameter "`cName`".

Parameter “`cADict`” is a dictionary. It is used for target individual generation. Dictionary key is the subelement tag and that attribute is set by “`oName`” parameter.

Method `addObjectFactWithAnnotation(self, path, oName, cName, cADict = {}, cSDict = {}, cANDict = {}, references = False, note = False)` extracts object facts with annotations. This is the most complex method. All parameters including “`cADict`” have the same purpose and meaning as in the above method.

Parameter “`cSDict`” is a dictionary used for extraction of data facts from subelements of some “`element`” subelements. The dictionary key is the subelement tag, and the dictionary value is the data fact name. Observed examples are more specific case because they may reference CVE individuals or simply to be `Reference` annotations.

Parameter “`cANDict`” is a dictionary. It used for annotation extraction for the target individual from attributes. Target individual is an individual that is generated. The object fact (object property) points to this target individual. The dictionary key is the subelement tag and the dictionary value is the annotation name. Annotations usually are added to the target individual, but in case of techniques and attack identifiers, annotations are added to the object facts of the target individual.

Method `addCWE(self)` adds referenced CWE individuals from attack patterns elements as object facts.

Method `addExcludeRelated(self, category)` adds excluded category ancestors of the attack. The ancestor category is given with parameter “`category`”.

Method `tostring(self)` serializes its object into a string containing individual description.

Method `addMembers(self, relationships = False)` is used to add views and categories. Link type is determined via “`relationships`” parameter – by default, category members are processed.

Method `addRelatedAttackPatterns(self)` adds the object facts for related attack patterns. Here, excluded categories are taken in account.

Method `addContent(self, capecID)` is specialized for `Has_Member` object fact with attack identifier for attack pattern. Parameter “`capecID`” contains this attack identifier.

Methods in `AttackPattern` class are combinations of with / without annotations, from attributes / subelements, from children / other descendants, for data / object properties. Here, only combinations that exist in CAPEC ontology and CAPEC XSD structure are implemented. This note is applicable for `Individual` class too.

### 2.2. Individual class

`Individual` class is a simplified implementation of `AttackPattern` class. It collects the characteristics of the target individuals of the object facts.

Here, class extend is supported (class variable “`extend`”), from which the description of all target individuals are serialized at the end of procedure `generateIndividuals (root)`.

Methods `addType(self, t)`, `addDataFact(self, d, v)`, `addAnnotation(self, a, v)`, `addObjectFact(self, d, v)`, `addObjectFactWithAnnotations(self, d, v, an, av)`, and `tostring(self)` are simplified implementations of the corresponding `AttackPattern` methods.

## 3. Conclusion

CAPEC generator implementation is similar to that of CWE generator. In both implementations, the control flow follows corresponding XSD schemas. Both implementations have two classes `AttachPattern`/`Weakness` and `Individual`. CWE generator classes are presented as UML class diagram in Figure 2.

From complier/translator point of view, ecosystem generators for CPE, CVE/NVD, CWE, and CAPEC ontologies are compilers from XML/JSON to OWL.

The components of every compiler are lexical, syntactical, semantical analyzers, and code generator. In this case, the analyzer functionality is derived from Python libraries. Only code generators are coded.

For CPE and CVE/NVD generators, concurrent implementations have been adapted to achieve acceptable production timings. In these implementations, all available cores and memory are used for processing.

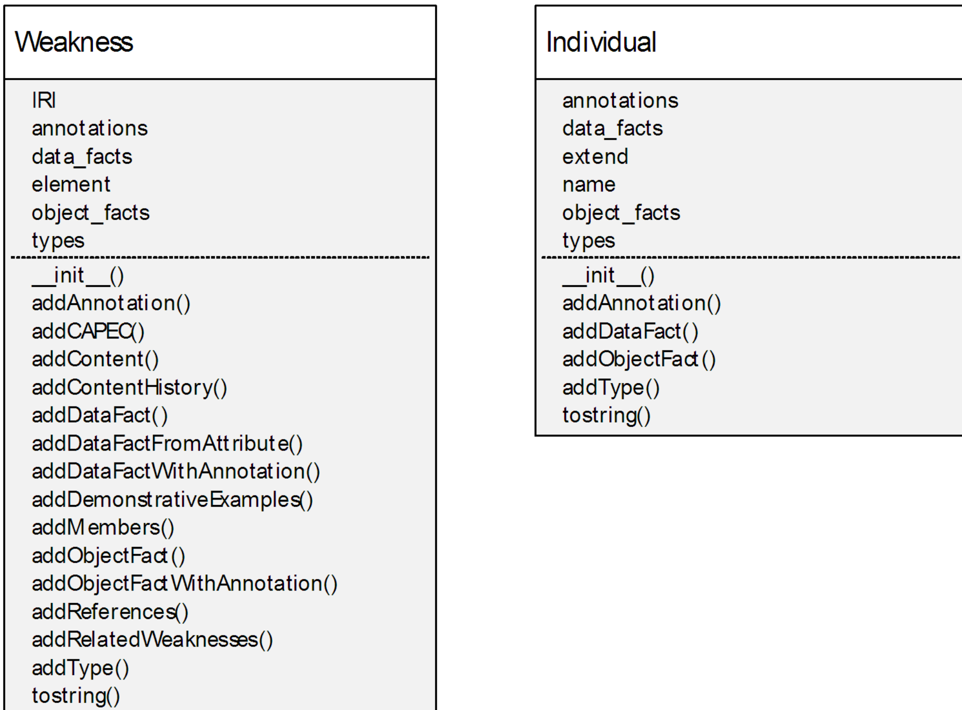


Figure 2. CWE generator classes

Many applications have no option to use available computer hardware resources. Usually, they work with one core and limited memory, independently of the job. This problem, here, in the above mentioned generators, is roundabout by manual programming. Today, we are still far away from full automatization of concurrent programming. Therefore, every change, especially in concurrent programs, is a difficult task.

Every change in XSD schemas reflect in generator implementations. This happened at least one time in the year.

Another problem is the change of generator data sources. For example, NIST plan in September 2023 to retire NVD and CPE data feed supplies and to start only JSON RESTfull services for that purpose. Therefore, the corresponding generators have to be rewritten to meet this change.

All ecosystem ontologies have been presented in OWL Manchester syntax [9]. This syntax is more human readable, but it is not used as input source for currently available triple stores. The last ones support Turtle syntax [10]. Therefore, the generators have to be rewritten to generate in Turtle syntax. This process is going on and some generators available on published yet addresses generate their ontologies in Turtle syntax. From programming point of view, it is not dramatic change. Something more, generator codes get simplified.

Ontologies in Turtle syntax for presentation purpose can be easily converted into Manchester syntax using some tools like Protégé [6] or Robot [8].

Ecosystem ontologies must be available in a triple store if we want the ecosystem to be used. The most important task is to load and update ecosystem ontologies into some triple store. However, CVE/NVD ontology is really huge. Currently, ecosystem ontologies contain approximately eight billion triples. Experiment shows that these ontologies can be loaded in a triple store on modern desktop computer for two months.

Experiments have been done with BrightstarDB [3] and Apache Jena [1]. These triple stores have their advantages and disadvantages, but their common problem is the slow batch load. Both triple stores do not use all available hardware resources. Apache Jena is somehow better than BrightstarDB, but still is not a solution.

BraghtstarDB is well integrated in .NET environment. Automatically, OWL ontologies can be presented as C# classes and used for object-oriented development. At the same time, there is access with SPARQL to its ontologies, but that is all.

Apache Jena is a complex of many Semantic web tools, but their integration is achieved via Java programming. Therefore programming in Apache Jena is not very automated.

The problem with CVE/NVD ontology actualization can be solved following NIST proposition. Vulnerabilities are organized in chunks by years. Therefore, they can be loaded once. Then, NIST offers data feed for modified vulnerabilities. This patch can be applied to update quickly CVE/NVD content.

## Acknowledgements

This paper is prepared with the support of MIRACle: Mechatronics, Innovation, Robotics, Automation, Clean Technologies – Establishment and Development of a Center for Competence in Mechatronics and Clean Technologies – Laboratory Intelligent Urban Environment, funded by the Operational Program Science and Education for Smart Growth 2014–2020, Project BG 05M2OP001-1.002-0011.

## References

[1] Apache Jena, A free and open source Java framework for building Semantic Web and Linked Data applications, https://jena.apache.org.

[2] S. Barnum and A. Sethi, Attack patterns as a knowledge resource for building secure software, Cigital, Inc., 2007, https://capec.mitre.org/documents/Attack_Patterns-Knowing_Your_Enemies_in_Order_to_Defeat_Them-Paper.pdf.

[3] BrightstarDB, A native RDF database for the .NET platform, https://brightstardb.com.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: Elements of reusable object-oriented software, Boston, MA, Addison-Wesley, 1995.

[5] MITRE Corporation, CAPEC (Common Pattern Enumeration and Classification), 2023, http://capec.mitre.org.

[6] M. A. Musen, The Protégé project: A look back and a look forward, AI Matters 1(4) (2015) 4–12, https://doi.org/10.1145/2757001.2757003.

[7] NIST, NVD (National Vulnerability Database), 2023, http://nvd.nist.gov.

[8] Open Biological and Biomedical Ontology Foundry (OBO), ROBOT, ROBOT is an OBO Tool, http://robot.obolibrary.org.

[9] W3C, OWL 2 Web Ontology Language, Manchester Syntax (Second Edition), W3C Working Group Note, 11 December 2012, https://www.w3.org/TR/owl2-manchester-syntax.

[10] W3C, RDF 1.1 Turtle, Terse RDF Triple Language, W3C Recommendation, 25 February 2014, https://www.w3.org/TR/turtle.

[11] W3C, Shapes Constraint Language (SHACL), W3C Recommendation, 20 July 2017, http://www.w3.org/TR/shacl.

[12] W3C, Semantic Web, Web Ontology Language (OWL), 2023, https://www.w3.org/OWL.

*Received on January 3, 2024*
*Accepted on February 27, 2024*

Vladimir Dimitrov

Faculty of Mathematics and Informatics
Sofia University "St. Kliment Ohridski"
5, James Bourchier Blvd.
1164 Sofia
BULGARIA

E-mail: cht@fmi.uni-sofia.bg