VulnerableCode

nexB Inc. and others

Nov 30, 2023

GETTING STARTED

1 VulnerableCode Overview 3
- 1.1 What can I do with VulnerableCode? 3
- 1.2 Why VulnerableCode? 3
- 1.3 How does it work? 4
- 1.4 How can I contribute to VulnerableCode? 4

2 User Interface 5
- 2.1 Search by packages 5
- 2.2 Search by vulnerabilities 6

3 Installation 11
- 3.1 Run with Docker 11
- 3.2 Local development installation 12
- 3.3 Using Nix 14

4 API overview 17
- 4.1 Browse the Open API documentation 17
- 4.2 How to use OpenAPI documentation 17
- 4.3 Query for Package Vulnerabilities 17
- 4.4 Package Bulk Search 18
- 4.5 CPE Bulk Search 18
- 4.6 API endpoints reference 19
- 4.7 Miscellaneous 21

5 API usage administration for on-premise deployments 23
- 5.1 Enable the API key authentication 23
- 5.2 Create an API key-only user 23

6 Contributing to VulnerableCode 25
- 6.1 Do Your Homework 25
- 6.2 Ways to Contribute 25
- 6.3 Helpful Resources 26
- 6.4 Add a new importer 26
- 6.5 TL;DR 27
- 6.6 Prerequisites 27
- 6.7 Writing an importer 28
- 6.8 Add a new improver 32
- 6.9 TL;DR 32
- 6.10 Prerequisites 32
- 6.11 Writing an improver 33

7 FAQ **37**
7.1 During development, how do I quickly empty my database and start afresh ? 37

8 Miscellaneous **39**
8.1 Continuous periodic Data import 39
8.2 Environment variables configuration 40
8.3 Throttling rate configuration 40

9 Importer Overview **41**

10 Improver Overview **43**

11 Framework Overview **45**

12 Command Line Interface **47**
12.1 `$ ./manage.py --help` 47
12.2 `$ ./manage.py <subcommand> --help` 47
12.3 `$ ./manage.py import <importer-name>` 48
12.4 `$ ./manage.py improve <improver-name>` 48
12.5 `$ ./manage.py purl2cpe --destination <directory` 48

13 Importers **49**

14 Google Summer of Code 2021 Final Report **51**
14.1 Organization - AboutCode 51
14.2 Overview 51
14.3 Detailed Report 51
14.4 Pre GSoC 53
14.5 Post GSoC - Future Plans and what's left 54
14.6 Closing Thoughts 54

15 Indices and tables **55**

VulnerableCode provides an open database of software packages that are affected by known security vulnerabilities aka. *"vulnerable packages"*.

VulnerableCode is also a free and open source software (FOSS) project that provides the tools to build this open database. The tools handle collecting, aggregating and correlating these vulnerabilities and relating them to a correct package version. Our project also supports a public cloud instance of this database - VulnerableCode.io.

In this documentation you will find information on:

- An overview of VulnerableCode and what you can do with it
- Installation instructions
- How to make technical contributions to the project and the community

CHAPTER

ONE

VULNERABLECODE OVERVIEW

VulnerableCode provides an open database of software packages that are affected by known security vulnerabilities aka *"vulnerable packages"*.

VulnerableCode is also a free and open source software (FOSS) project that provides the tools to build this open database. The tools handle collecting, aggregating and correlating these vulnerabilities and relating them to a correct package version. Our project also supports a public cloud instance of this database - VulnerableCode.io.

1.1 What can I do with VulnerableCode?

For security researchers and software developers, VulnerableCode offers a web UI and a JSON API to efficiently find if the FOSS packages and dependencies that you use are affected by known vulnerabilities and to determine whether a later package version fixes those vulnerabilities.

- With the web UI, you can search by package using Package URLs or search by vulnerability, e.g., by CVE. From there you can navigate to the package vulnerabilities and to the vulnerable packages.
- With the JSON API, you can perform package queries using Package URLs (purl) or query by vulnerability id ("VCID"). You can also query by CPEs and other vulnerability aliases. The API provides paginated index and detail endpoints and includes indexes of vulnerable CPEs and vulnerable Package URLs.

You can install VulnerableCode locally or use the provided publicly hosted instance, or host your own installation. You can contact the VulnerableCode team for special needs including commercial support.

1.2 Why VulnerableCode?

VulnerableCode provides open correlated data and will support curated data. Our approach is to prioritize upstream data sources and to merge multiple vulnerability data sources after comparison and correlation. The vulnerability data is keyed by Package URL ensuring quick and accurate lookup with minimal friction. We continuously validate and refine the collected data for quality, accuracy and consistency using "improver" jobs. An example is an improver that can validate that a package version reported as vulnerable actually exists (some do not exist). Another example is to re-evaluate vulnerable version ranges based on the latest releases of a package.

The benefit of our approach is that we will eventually provide better, more accurate vulnerability data for packages reported in an SBOM. This should contribute to more efficient vulnerability management with less noise from false positives.

Another key reason why we created VulnerableCode is that existing vulnerability database solutions are primarily commercial or proprietary. This does not make sense because the bulk of the vulnerability data is about FOSS.

The National Vulnerability Database, which is a primary centralized data source for known vulnerabilities, is not particularly well suited to address FOSS security issues because:

1. It predates the explosion of FOSS software usage
2. Its data format reflects a commercial vendor-centric point of view in part due to the usage of CPEs to map vulnerabilities to existing packages.
3. CPEs were not designed to map FOSS to vulnerabilities owing to their vendor-product centric semantics. This makes it really hard to answer the fundamental questions: "Is package *foo* vulnerable?" and "Is package *foo* vulnerable to vulnerability *bar*?"

1.3 How does it work?

VulnerableCode independently aggregates many software vulnerability data sources and supports data re-creation in a decentralized fashion. These data sources (see complete list here) include security advisories published by Linux and BSD distributions, application software package managers and package repositories, FOSS projects, GitHub and more. Thanks to this approach, the data is focused on specific ecosystems and aggregated in a single database that enables querying a richer graph of relations between multiple representations of a package. Being specific increases the accuracy and validity of the data as the same version of an upstream package across different ecosystems may or may not be subject to the same vulnerability.

In VulnerableCode, packages are identified using Package URL (purl) as the primary identifier instead of a CPE. This makes answers to questions such as "Is package *foo* vulnerable to vulnerability *bar*?" more accurate and easier to interpret.

The primary access to VulnerableCode data is through a REST API, but there is also a Web UI for searching and browsing vulnerabilities by package or by vulnerability. For the initial releases both access modes are read-only, but our longer-term goal is to enable community curation of the data including addition of new packages and vulnerabilities, and reviewing and updating their relationships.

We also plan to mine for vulnerabilities that have not received any exposure due to reasons such as the complicated procedure to obtain a CVE ID or not being able to classify a bug as a vulnerability.

1.4 How can I contribute to VulnerableCode?

Please get in touch on our Gitter channel. You can review or get the code and report issues at our GitHub repo.

CHAPTER

TWO

USER INTERFACE

2.1 Search by packages

The search by packages is a very powerful feature of VulnerableCode. It allows you to search for packages by the package URL or purl prefix fragment such as `pkg:pypi` or by package name.

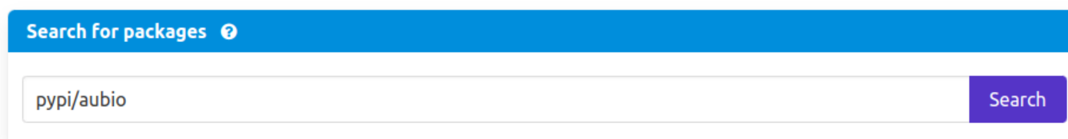
The search by packages is available at the following URL:

> https://public.vulnerablecode.io/packages/search

How to search by packages:

1. Go to the URL: https://public.vulnerablecode.io/packages/search
2. Enter the package URL or purl prefix fragment such as `pkg:pypi` or by package name in the search box.
3. Click on the search button.

The search results will be displayed in the table below the search box.



11 results

Package URL	Affected by vulnerabilities	Fixing vulnerabilities
pkg:pypi/aubio@0.4.3a1	9	0
pkg:pypi/aubio@0.4.3a2	9	0
pkg:pypi/aubio@0.4.3	9	0
pkg:pypi/aubio@0.4.3.post1	9	0
pkg:pypi/aubio@0.4.4	9	0
pkg:pypi/aubio@0.4.5	9	0
pkg:pypi/aubio@0.4.6	9	0
pkg:pypi/aubio@0.4.7	3	6
pkg:pypi/aubio@0.4.0	3	0
pkg:pypi/aubio@0.4.9	0	3
pkg:pypi/aubio@0.4.8	3	0

Click on the package URL to view the package details.

Search for packages

pypi/audio

Search

Package details:

purl	pkg:pypi/aubio@0.4.7

Affected by vulnerabilities (3)

Vulnerability	Summary	Aliases
VCID-dxr5-n98h-aaaf	aubio has a Buffer Overflow vulnerability in `new_aubio_tempo`	CVE-2018-19800 GHSA-grmf-4fq6-2r79 PYSEC-2019-162
VCID-ebsp-fdzq-aaac	NULL Pointer Dereference aubio has a `new_aubio_onset` NULL pointer dereference.	CVE-2018-19802 GHSA-c6jq-h4jp-72pr PYSEC-2019-164
VCID-rjc1-h55r-aaae	aubio v0.4.0 to v0.4.8 has a NULL pointer dereference in new_aubio_filterbank via invalid n_filters.	CVE-2018-19801 GHSA-7vvr-h4p5-m7fh PYSEC-2019-163

Fixing vulnerabilities (6)

Vulnerability	Summary	Aliases
VCID-2x24-vevc-aaag	An issue was discovered in aubio 0.4.6. A buffer over-read can occur in new_aubio_pitchyinfft in pitch/pitchyinfft.c, as demonstrated by aubionotes.	CVE-2018-14523 PYSEC-2018-63
VCID-6sx4-nwbk-aaas	Improper Restriction of Operations within the Bounds of a Memory Buffer An issue was discovered in aubio. A `SEGV` signal can occur in `aubio_source_avcodec_readframe` in `io/source_avcodec.c`, as demonstrated by `aubiomfcc`.	CVE-2018-14521 PYSEC-2018-61
VCID-hjbg-pjg6-aaaj	The swri_audio_convert function in audioconvert.c in FFmpeg libswresample through 3.0.101, as used in FFmpeg 3.4.1, aubio	CVE-2017-17555 PYSEC-2017-77

2.2 Search by vulnerabilities

The search by vulnerabilities is a very powerful feature of VulnerableCode. It allows you to search for vulnerabilities by the VCID itself. It also allows you to search for vulnerabilities by the CVE, GHSA, CPEs etc or by the fragment of these identifiers like `CVE-2021`.

The search by vulnerabilities is available at the following URL:

> https://public.vulnerablecode.io/vulnerabilities/search

How to search by vulnerabilities:

1. Go to the URL: https://public.vulnerablecode.io/vulnerabilities/search
2. Enter the VCID, CVE, GHSA, CPEs etc. in the search box.
3. Click on the search button.

The search results will be displayed in the table below the search box.

Search for vulnerabilities

CVE-2021 Search

20,860 results Previous 1 … 8 … 1043 Next

Vulnerability id	Aliases	Affected packages	Fixed by packages
VCID-17gr-j9w6-aaan	CVE-2021-24493	0	0
VCID-17hu-m7ks-aaad	CVE-2021-41535	0	0
VCID-17kk-7ps2-aaaq	CVE-2021-29621 GHSA-434h-p4gx-jm89 PYSEC-2021-90	238	12
VCID-17nu-1tve-aaae	CVE-2021-1435	0	0
VCID-17pc-vrrp-aaab	CVE-2021-24744	0	0
VCID-17r7-8esr-aaaj	CVE-2021-3682	4	39
VCID-17rt-bgxy-aaac	CVE-2021-0395	0	0
VCID-17rv-78vf-aaap	CVE-2021-28575	0	0
VCID-17s9-gfhk-aaab	CVE-2021-43925	0	0
VCID-17tm-xhzs-aaag	CVE-2021-21687 GHSA-3q84-vrvx-rfvf	7	1
VCID-17uf-xv76-aaae	CVE-2021-33923	0	0
VCID-17xc-xbpv-aaah	CVE-2021-36870	0	0
VCID-182a-697u-aaaf	CVE-2021-41504	0	0
VCID-182k-m14n-aaae	CVE-2021-25810	0	0
VCID-1842-18v3-aaab	CVE-2021-28976	0	0
VCID-18ap-rajg-aaan	CVE-2021-3688	26	0
VCID-18c6-cbhg-aaag	CVE-2021-42326	0	15
VCID-18dr-y2pm-aaad	CVE-2021-21846	0	23
VCID-18fm-tq9g-aaan	CVE-2021-1173	0	0
VCID-18fq-3zkc-aaap	CVE-2021-45757	0	0

Click on the VCID to view the vulnerability details.

Vulnerability details: VCID-17tm-xhzs-aaag

Essentials | Fixed by packages (1) | Affected packages (7) | References (12)

Vulnerability ID	VCID-17tm-xhzs-aaag
Aliases	CVE-2021-21687 GHSA-3q84-vrvx-rfvf
Summary	Missing Authorization Jenkins does not check agent-to-controller access to create symbolic links when unarchiving a symbolic link in FilePath#untar.

Severity (10)

System	Score	Found at
cvssv3	9.0	https://access.redhat.com/hydra/rest/securitydata/cve/CVE-2021-21687.json
rhbs	high	https://bugzilla.redhat.com/show_bug.cgi?id=2020324
cvssv2	6.4	https://nvd.nist.gov/vuln/detail/CVE-2021-21687
cvssv3	9.1	https://nvd.nist.gov/vuln/detail/CVE-2021-21687
cvssv3.1_qr	CRITICAL	https://github.com/advisories/GHSA-3q84-vrvx-rfvf
rhas	Important	https://access.redhat.com/errata/RHSA-2021:4799
rhas	Important	https://access.redhat.com/errata/RHSA-2021:4801
rhas	Important	https://access.redhat.com/errata/RHSA-2021:4827
rhas	Important	https://access.redhat.com/errata/RHSA-2021:4829
rhas	Important	https://access.redhat.com/errata/RHSA-2021:4833

Fixed by packages (1)

pkg:maven/org.jenkins-ci.main/jenkins-core@2.319

Affected packages (7)

pkg:maven/org.jenkins-ci.main/jenkins-core@2.304

pkg:maven/org.jenkins-ci.main/jenkins-core@2.318

Affected packages tab shows the list of packages affected by the vulnerability.

Vulnerability details: VCID-17tm-xhzs-aaag

Essentials | Fixed by packages (1) | **Affected packages (7)** | References (12)

Package URL
pkg:maven/org.jenkins-ci.main/jenkins-core@2.304
pkg:maven/org.jenkins-ci.main/jenkins-core@2.318
pkg:rpm/redhat/jenkins@2.303.3.1637595827-1?arch=el8
pkg:rpm/redhat/jenkins@2.303.3.1637596565-1?arch=el8
pkg:rpm/redhat/jenkins@2.303.3.1637597018-1?arch=el8
pkg:rpm/redhat/jenkins@2.303.3.1637597493-1?arch=el8
pkg:rpm/redhat/jenkins@2.303.3.1637698110-1?arch=el7

Fixed by packages tab shows the list of packages that fix the vulnerability.

Vulnerability details: VCID-17tm-xhzs-aaag

Essentials | **Fixed by packages (1)** | Affected packages (7) | References (12)

Package URL
pkg:maven/org.jenkins-ci.main/jenkins-core@2.319

CHAPTER THREE

INSTALLATION

Warning: VulnerableCode is going through a major structural change and the installations are likely to not produce enough results. This is tracked in https://github.com/nexB/vulnerablecode/issues/597

Welcome to **VulnerableCode** installation guide! This guide describes how to install VulnerableCode on various platforms. Please read and follow the instructions carefully to ensure your installation is functional and smooth.

The **preferred VulnerableCode installation** is to *Run with Docker* as this is the simplest to setup and get started. Running VulnerableCode with Docker **guarantees the availability of all features** with the **minimum configuration** required. This installation **works across all Operating Systems**.

Alternatively, you can install VulnerableCode locally as a development server with some limitations and caveats. Refer to the *Local development installation* section.

3.1 Run with Docker

3.1.1 Get Docker

The first step is to download and **install Docker on your platform**. Refer to Docker documentation and choose the best installation path for your system: Get Docker.

3.1.2 Build the Image

VulnerableCode is distributed with `Dockerfile` and `docker-compose.yml` files required for the creation of the Docker image.

Clone the git VulnerableCode repo, create an environment file, and build the Docker image:

```
git clone https://github.com/nexB/vulnerablecode.git && cd vulnerablecode
make envfile
docker-compose build
```

Note: The image will need to be re-built when the VulnerableCode app source code is modified or updated via `docker-compose build --no-cache vulnerablecode`

3.1.3 Run the App

Run your image as a container:

```
docker-compose up
```

At this point, the VulnerableCode app should be running at port `8000` on your Docker host. Go to http://localhost:8000/ on a web browser to access the web UI. Optionally, you can set `NGINX_PORT` environment variable in your shell or in the *.env* file to run on a different port than 8000.

Note: To access a dockerized VulnerableCode app from a remote location, the `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` setting need to be provided in your `docker.env` file:

```
ALLOWED_HOSTS=.domain.com,127.0.0.1
CSRF_TRUSTED_ORIGINS=https://*.domain.com,http://127.0.0.1
```

Refer to Django ALLOWED_HOSTS settings and CSRF_TRUSTED_ORIGINS settings for more details.

Warning: Serving VulnerableCode on a network could lead to security issues and there are several steps that may be needed to secure such a deployment. Currently, this is not recommendend.

3.1.4 Execute a Command

You can execute a one of `manage.py` commands through the Docker command line interface, for example:

```
docker-compose run vulnerablecode ./manage.py import --list
```

Note: Refer to the *Command Line Interface* section for the full list of commands.

Alternatively, you can connect to the Docker container `bash` and run commands from there:

```
docker-compose run vulnerablecode bash
./manage.py import --list
```

3.2 Local development installation

3.2.1 Supported Platforms

VulnerableCode has been tested and is supported on the following operating systems:

1. **Debian-based** Linux distributions
2. **macOS** 12.1 and up

Warning: On **Windows** VulnerableCode can **only** *Run with Docker* and is not supported.

3.2.2 Pre-installation Checklist

Before you install VulnerableCode, make sure you have the following prerequisites:

- **Python: 3.8+** found at https://www.python.org/downloads/
- **Git**: most recent release available at https://git-scm.com/
- **PostgreSQL**: release 10 or later found at https://www.postgresql.org/ or https://postgresapp.com/ on macOS

3.2.3 System Dependencies

In addition to the above pre-installation checklist, there might be some OS-specific system packages that need to be installed before installing VulnerableCode.

On **Debian-based distros**, several **system packages are required** by VulnerableCode. Make sure those are installed:

```
sudo apt-get install python3-venv python3-dev postgresql libpq-dev build-essential
```

3.2.4 Clone and Configure

Clone the VulnerableCode Git repository:

```
git clone https://github.com/nexB/vulnerablecode.git && cd vulnerablecode
```

Install the required dependencies:

```
make dev
```

Note: You can specify the Python version during the `make dev` step using the following command:

```
make dev PYTHON_EXE=python3.8.10
```

When `PYTHON_EXE` is not specified, by default, the `python3` executable is used.

Create an environment file:

```
make envfile
```

3.2.5 Database

PostgreSQL is the preferred database backend and should always be used on production servers.

- Create the PostgreSQL user, database, and table with:

  ```
  make postgres
  ```

Note: You can also use a **SQLite** database for local development as a single user with:

```
make sqlite
```

Warning: Choosing SQLite over PostgreSQL has some caveats. Check this link for more details.

3.2.6 Tests

You can validate your VulnerableCode installation by running the tests suite:

```
make test
```

3.2.7 Web Application

A web application is available to create and manage your projects from a browser; you can start the local webserver and access the app with:

```
make run
```

Then open your web browser and visit: http://127.0.0.1:8000/ to access the web application.

Warning: This setup is **not suitable for deployments** and **only supported for local development**.

3.2.8 Upgrading

If you already have the VulnerableCode repo cloned, you can upgrade to the latest version with:

```
cd vulnerablecode
git pull
make dev
make migrate
```

3.3 Using Nix

You can install VulnerableCode with Nix (Flake support is needed):

```
cd etc/nix
nix-shell -p nixFlakes --run "nix --print-build-logs flake check " # build & run tests
```

There are several options to use the Nix version:

```
# Enter an interactive environment with all dependencies set up.
cd etc/nix
nix develop
> ../../manage.py ... # invoke the local checkout
> vulnerablecode-manage.py ... # invoke manage.py as installed in the nix store

# Test the import prodecure using the Nix version.
```

(continues on next page)

(continued from previous page)

```
etc/nix/test-import-using-nix.sh --all # import everything
# Test the import using the local checkout.
INSTALL_DIR=. etc/nix/test-import-using-nix.sh ruby # import ruby only
```

Keeping the Nix setup in sync

The Nix installation uses mach-nix to handle Python dependencies because some dependencies are currently not available as Nix packages. All Python dependencies are automatically fetched from `./requirements.txt`. If the `mach-nix`-based installation fails, you might need to update `mach-nix` itself and the pypi-deps-db version in use (see `etc/nix/flake.nix:inputs.machnix` and `machnixFor.pypiDataRev`).

Non-Python dependencies are curated in:

```
etc/nix/flake.nix:vulnerablecode.propagatedBuildInputs
```

CHAPTER

FOUR

API OVERVIEW

4.1 Browse the Open API documentation

- https://public.vulnerablecode.io/api/docs/ for documentation with Swagger
- https://public.vulnerablecode.io/api/schema/ for the OpenAPI schema

4.2 How to use OpenAPI documentation

The API documentation is available at https://public.vulnerablecode.io/api/docs/. To use the endpoints you need to authenticate with an API key. Request your API key from https://public.vulnerablecode.io/account/request_api_key/. Once you have your API key, click on the `Authorize` button on the top right of the page and enter your API key in the `value` field with `Token` prefix, so if your token is "1234567890abcdef" then you have to enter this: `Token 1234567890abcdef`.

4.3 Query for Package Vulnerabilities

The package endpoint allows you to query vulnerabilities by package using a purl or purl fields.

Sample python script:

```
import requests

# Query by purl
resp = requests.get(
    "https://public.vulnerablecode.io/api/packages?purl=pkg:maven/log4j/log4j@1.2.27",
    headers={"Authorization": "Token 123456789"},
).json()

# Query by purl type, get all the vulnerable maven packages
resp = requests.get(
    "https://public.vulnerablecode.io/api/packages?type=maven",
    headers={"Authorization": "Token 123456789"},
).json()
```

Sample using curl:

```
curl -X GET -H 'Authorization: Token <YOUR TOKEN>' https://public.vulnerablecode.io/api/
↪packages?purl=pkg:maven/log4j/log4j@1.2.27
```

The response will be a list of packages, these are packages that are affected by and/or that fix a vulnerability.

4.4 Package Bulk Search

The package bulk search endpoint allows you to search for purls in bulk. You can pass a list of purls in the request body and the endpoint will return a list of purls with vulnerabilities.

You can pass a list of `purls` in the request body. Each package should be a valid purl string.

You can also pass options like `purl_only` and `plain_purl` in the request. `purl_only` will return only a list of vulnerable purls from the purls received in request. `plain_purl` allows you to query the API using plain purls by removing qualifiers and subpath from the purl.

The request body should be a JSON object with the following structure:

```
{
    "purls": [
        "pkg:pypi/flask@1.2.0",
        "pkg:npm/express@1.0"
    ],
    "purl_only": false,
    "plain_purl": false,
}
```

Sample python script:

```
import requests

request_body = {
    "purls": [
        "pkg:npm/grunt-radical@0.0.14"
    ],
}

resp = requests.post('https://public.vulnerablecode.io/api/packages/bulk_search', json=␣
↪request_body, headers={'Authorization': "Token 123456789"}).json()
```

The response will be a list of packages, these are packages that are affected by and/or that fix a vulnerability.

4.5 CPE Bulk Search

The CPE bulk search endpoint allows you to search for packages in bulk. You can pass a list of packages in the request body and the endpoint will return a list of vulnerabilities.

You can pass a list of `cpes` in the request body. Each cpe should be a non empty string and a valid CPE.

The request body should be a JSON object with the following structure:

```
{
    "cpes": [
        "cpe:2.3:a:apache:struts:2.3.1:*:*:*:*:*:*:*",
        "cpe:2.3:a:apache:struts:2.3.2:*:*:*:*:*:*:*"
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Sample python script:

```
import requests

request_body = {
    "cpes": [
        "cpe:2.3:a:apache:struts:2.3.1:*:*:*:*:*:*:*"
    ],
}

resp = requests.post('https://public.vulnerablecode.io/api/cpes/bulk_search', json=␣
↪request_body, headers={'Authorization': "Token 123456789"}).json()
```

The response will be a list of vulnerabilities that have the following CPEs.

4.6 API endpoints reference

There are two primary endpoints:

- packages/: this is the main endpoint where you can lookup vulnerabilities by package.
- vulnerabilities/: to lookup by vulnerabilities

And two secondary endpoints, used to query vulnerability aliases (such as CVEs) and vulnerability by CPEs: cpes/ and aliases/

Table 1: Table for the main API endpoints

Endpoint	Query Parameters	Expected Output
/api/packages	• purl (string) = package-url of the package • type (string) = type of the package • namespace (string) = namespace of the package • name (string) = name of the package • version (string) = version of the package • qualifiers (string) = qualifiers of the package • subpath (string) = subpath of the package • page (integer) = page number of the response • page_size (integer) = number of packages in each page	Return a list of packages using a package-url (purl) or a combination of type, namespace, name, version, qualifiers, subpath purl fields. See the purl specification for more details. See example at *Query for Package Vulnerabilities* section for more details.
/api/packages/bulk_search	Refer to package bulk search section *Package Bulk Search*	Return a list of packages
/api/vulnerabilities/	• vulnerability_id (string) = VCID (VulnerableCode Identifier) of the vulnerability • page (integer) = page number of the response • page_size (integer) = number of vulnerabilities in each page	Return a list of vulnerabilities
/api/cpes	• cpe (string) = value of the cpe • page (integer) = page number of the response • page_size (integer) = number of cpes in each page	Return a list of vulnerabilities
/api/cpes/bulk_search	Refer to CPE bulk search section *CPE Bulk Search*	Return a list of cpes
/api/aliases	• alias (string) = value of the alias • page (integer) = page number of the response • page_size (integer) = number of aliases in each page	Return a list of vulnerabilities

Table 2: Table for other API endpoints

Endpoint	Query Parameters	Expected Output
`/api/packages/{id}`	• `id` (integer) = internal primary id of the package	Return a package with the given id
`/api/packages/all`	No parameter required	Return a list of all vulnerable packages
`/api/vulnerabilities/{id}`	• `id` (integer) = internal primary id of the vulnerability	Return a vulnerability with the given id
`/api/aliases/{id}`	• `id` (integer) = internal primary id of the alias	Return an alias with the given id
`/api/cpes/{id}`	• `id` = internal primary id of the cpe	Return a cpe with the given id

4.7 Miscellaneous

The API is paginated and the default page size is 100. You can change the page size by passing the `page_size` parameter. You can also change the page number by passing the `page` parameter.

CHAPTER

FIVE

API USAGE ADMINISTRATION FOR ON-PREMISE DEPLOYMENTS

5.1 Enable the API key authentication

There is a setting VULNERABLECODEIO_REQUIRE_AUTHENTICATION for this. Use it this way:

```
$ VULNERABLECODEIO_REQUIRE_AUTHENTICATION=1 make run
```

5.2 Create an API key-only user

This can be done in the admin and from the command line:

```
$ ./manage.py create_api_user --email "p4@nexb.com" --first-name="Phil" --last-name "Goel
↪"
User p4@nexb.com created with API key: ce8616b929d2adsddd6146346c2f26536423423491
```

CHAPTER

SIX

CONTRIBUTING TO VULNERABLECODE

Thank you so much for being so interested in contributing to VulnerableCode. We are always on the lookout for enthusiastic contributors like you who can make our project better, and we are willing to lend a helping hand if you have any questions or need guidance along the way. That being said, here are a few resources to help you get started.

Note: By contributing to the VulnerableCode project, you agree to the Developer Certificate of Origin.

6.1 Do Your Homework

Before adding a contribution or create a new issue, take a look at the project's README, read through our documentation, and browse existing issues, to develop some understanding of the project and confirm whether a given issue/feature has previously been discussed.

6.2 Ways to Contribute

Contributing to the codebase is not the only way to add value to VulnerableCode or join our community. Below are some examples to get involved:

6.2.1 First Timers

You are here to help, but you are a new contributor! No worries, we always welcome newcomer contributors. We maintain some good first issues and encourage new contributors to work on those issues for a smooth start.

Tip: If you are an open-source newbie, make sure to check the extra resources at the bottom of this page to get the hang of the contribution process!

6.2.2 Code Contributions

For more established contributors, you can contribute to the codebase in several ways:

- Report a bug; just remember to be as specific as possible.
- Submit a bug fix for any existing issue.
- Create a new issue to request a feature, submit a feedback, or ask a question.

Note: Make sure to check existing issues, to confirm whether a given issue or a question has previously been discussed.

6.2.3 Documentation Improvements

Documentation is a critical aspect of any project that is usually neglected or overlooked. We value any suggestions to improve vulnerablecode documentation.

Tip: Our documentation is treated like code. Make sure to check our writing guidelines to help guide new users.

6.2.4 Other Ways

You want to contribute to other aspects of the VulnerableCode project, and you cannot find what you are looking for! You can always discuss new topics, ask questions, and interact with us and other community members on AboutCode Gitter and VulnerableCode Gitter

6.3 Helpful Resources

- Review our comprehensive guide for more details on how to add quality contributions to our codebase and documentation
- Check this free resource on how to contribute to an open source project on github
- Follow this wiki page on how to write good commit messages
- Pro Git book
- How to write a good bug report

6.4 Add a new importer

This tutorial contains all the things one should know to quickly implement an importer. Many internal details about importers can be found inside the `vulnerabilites/importer.py` file. Make sure to go through *Importer Overview* before you begin writing one.

6.5 TL;DR

1. Create a new `vulnerabilities/importers/importer_name.py` file.
2. Create a new importer subclass inheriting from the `Importer` superclass defined in `vulnerabilites.importer`. It is conventional to end an importer name with *Importer*.
3. Specify the importer license.
4. Implement the `advisory_data` method to process the data source you are writing an importer for.
5. Add the newly created importer to the importers registry at `vulnerabilites/importers/__init__.py`

6.6 Prerequisites

Before writing an importer, it is important to familiarize yourself with the following concepts.

6.6.1 PackageURL

VulnerableCode extensively uses Package URLs to identify a package. See the PackageURL specification and its Python implementation for more details.

Example usage:

```
from packageurl import PackageURL
purl = PackageURL(name="ffmpeg", type="deb", version="1.2.3")
```

6.6.2 AdvisoryData

`AdvisoryData` is an intermediate data format: it is expected that your importer will convert the raw scraped data into `AdvisoryData` objects. All the fields in `AdvisoryData` dataclass are optional; it is the importer's resposibility to ensure that it contains meaningful information about a vulnerability.

6.6.3 AffectedPackage

`AffectedPackage` data type is used to store a range of affected versions and a fixed version of a given package. For all version-related data, univers library is used.

6.6.4 Univers

univers is a Python implementation of the vers specification. It can parse and compare all the package versions and all the ranges, from debian, npm, pypi, ruby and more. It processes all the version range specs and expressions.

6.6.5 Importer

All the generic importers need to implement the `Importer` class. For `Git` or `Oval` data source, `GitImporter` or `OvalImporter` could be implemented.

Note: `GitImporter` and `OvalImporter` need a complete rewrite. Interested in *Contributing to VulnerableCode* ?

6.7 Writing an importer

6.7.1 Create Importer Source File

All importers are located in the `vulnerabilites/importers` directory. Create a new file to put your importer code in. Generic importers are implemented by writing a subclass for the `Importer` superclass and implementing the unimplemented methods.

6.7.2 Specify the Importer License

Importers scrape data off the internet. In order to make sure the data is useable, a license must be provided. Populate the `spdx_license_expression` with the appropriate value. The SPDX license identifiers can be found at https://spdx.org/licenses/.

Note: An SPDX license identifier by itself is a valid licence expression. In case you need more complex expressions, see https://spdx.github.io/spdx-spec/v2.3/SPDX-license-expressions/

6.7.3 Implement the `advisory_data` Method

The `advisory_data` method scrapes the advisories from the data source this importer is targeted at. It is required to return an *Iterable of AdvisoryData objects*, and thus it is a good idea to yield from this method after creating each AdvisoryData object.

At this point, an example importer will look like this:

`vulnerabilites/importers/example.py`

```
from typing import Iterable

from packageurl import PackageURL

from vulnerabilities.importer import AdvisoryData
from vulnerabilities.importer import Importer


class ExampleImporter(Importer):

    spdx_license_expression = "BSD-2-Clause"

    def advisory_data(self) -> Iterable[AdvisoryData]:
        return []
```

This importer is only a valid skeleton and does not import anything at all.

Let us implement another dummy importer that actually imports some data.

Here we have a `dummy_package` which follows `NginxVersionRange` and `SemverVersion` for version management from univers.

Note: It is possible that the versioning scheme you are targeting has not yet been implemented in the univers library. If this is the case, you will need to head over there and implement one.

```
from datetime import datetime
from datetime import timezone
from typing import Iterable

import requests
from packageurl import PackageURL
from univers.version_range import NginxVersionRange
from univers.versions import SemverVersion

from vulnerabilities.importer import AdvisoryData
from vulnerabilities.importer import AffectedPackage
from vulnerabilities.importer import Importer
from vulnerabilities.importer import Reference
from vulnerabilities.importer import VulnerabilitySeverity
from vulnerabilities.severity_systems import SCORING_SYSTEMS


class ExampleImporter(Importer):

    spdx_license_expression = "BSD-2-Clause"

    def advisory_data(self) -> Iterable[AdvisoryData]:
        raw_data = fetch_advisory_data()
        for data in raw_data:
            yield parse_advisory_data(data)


def fetch_advisory_data():
    return [
        {
            "id": "CVE-2021-23017",
            "summary": "1-byte memory overwrite in resolver",
            "advisory_severity": "medium",
            "vulnerable": "0.6.18-1.20.0",
            "fixed": "1.20.1",
            "reference": "http://mailman.nginx.org/pipermail/nginx-announce/2021/000300.
↪html",
            "published_on": "14-02-2021 UTC",
        },
        {
            "id": "CVE-2021-1234",
            "summary": "Dummy advisory",
            "advisory_severity": "high",
```

(continues on next page)

(continued from previous page)

```
            "vulnerable": "0.6.18-1.20.0",
            "fixed": "1.20.1",
            "reference": "http://example.com/cve-2021-1234",
            "published_on": "06-10-2021 UTC",
        },
    ]


def parse_advisory_data(raw_data) -> AdvisoryData:
    purl = PackageURL(type="example", name="dummy_package")
    affected_version_range = NginxVersionRange.from_native(raw_data["vulnerable"])
    fixed_version = SemverVersion(raw_data["fixed"])
    affected_package = AffectedPackage(
        package=purl, affected_version_range=affected_version_range, fixed_version=fixed_
↪version
    )
    severity = VulnerabilitySeverity(
        system=SCORING_SYSTEMS["generic_textual"], value=raw_data["advisory_severity"]
    )
    references = [Reference(url=raw_data["reference"], severities=[severity])]
    date_published = datetime.strptime(raw_data["published_on"], "%d-%m-%Y %Z").replace(
        tzinfo=timezone.utc
    )

    return AdvisoryData(
        aliases=[raw_data["id"]],
        summary=raw_data["summary"],
        affected_packages=[affected_package],
        references=references,
        date_published=date_published,
    )
```

Note:

Use `make valid` to format your new code using black and isort automatically.
Use `make check` to check for formatting errors.

6.7.4 Register the Importer

Finally, register your importer in the importer registry at `vulnerabilites/importers/__init__.py`

```
from vulnerabilities.importers import example
from vulnerabilities.importers import nginx

IMPORTERS_REGISTRY = [nginx.NginxImporter, example.ExampleImporter]

IMPORTERS_REGISTRY = {x.qualified_name: x for x in IMPORTERS_REGISTRY}
```

Congratulations! You have written your first importer.

6.7.5 Run Your First Importer

If everything went well, you will see your importer in the list of available importers.

```
$ ./manage.py import --list

Vulnerability data can be imported from the following importers:
vulnerabilities.importers.nginx.NginxImporter
vulnerabilities.importers.example.ExampleImporter
```

Now, run the importer.

```
$ ./manage.py import vulnerabilities.importers.example.ExampleImporter

Importing data using vulnerabilities.importers.example.ExampleImporter
Successfully imported data using vulnerabilities.importers.example.ExampleImporter
```

See *Command Line Interface* for command line usage instructions.

6.7.6 Enable Debug Logging (Optional)

For more visibility, turn on debug logs in `vulnerablecode/settings.py`.

```
DEBUG = True
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'root': {
        'handlers': ['console'],
        'level': 'DEBUG',
    },
}
```

Invoke the import command now and you will see (in a fresh database):

```
$ ./manage.py import vulnerabilities.importers.example.ExampleImporter

Importing data using vulnerabilities.importers.example.ExampleImporter
Starting import for vulnerabilities.importers.example.ExampleImporter
[*] New Advisory with aliases: ['CVE-2021-23017'], created_by: vulnerabilities.importers.
↪example.ExampleImporter
[*] New Advisory with aliases: ['CVE-2021-1234'], created_by: vulnerabilities.importers.
↪example.ExampleImporter
Finished import for vulnerabilities.importers.example.ExampleImporter. Imported 2␣
↪advisories.
Successfully imported data using vulnerabilities.importers.example.ExampleImporter
```

6.8 Add a new improver

This tutorial contains all the things one should know to quickly implement an improver. Many internal details about improvers can be found inside the `vulnerabilites/improver.py` file. Make sure to go through *Improver Overview* before you begin writing one.

6.9 TL;DR

1. Locate the importer that this improver will be improving data of at `vulnerabilities/importers/importer_name.py` file.
2. Create a new improver subclass inheriting from the `Improver` superclass defined in `vulnerabilites.improver`. It is conventional to end an improver name with *Improver*.
3. Implement the `interesting_advisories` property to return a QuerySet of imported data (`Advisory`) you are interested in.
4. Implement the `get_inferences` method to return an iterable of `Inference` objects for the given `AdvisoryData`.
5. Add the newly created improver to the improvers registry at `vulnerabilites/improvers/__init__.py`.

6.10 Prerequisites

Before writing an improver, it is important to familiarize yourself with the following concepts.

6.10.1 Importer

Importers are responsible for scraping vulnerability data from various data sources without creating a complete relational model between vulnerabilites and their fixes and storing them in a structured fashion. These data are stored in the `Advisory` model and can be converted to an equivalent `AdvisoryData` for various use cases. See *Importer Overview* for a brief overview on importers.

6.10.2 Importer Prerequisites

Improvers consume data produced by importers, and thus it is important to familiarize yourself with *Importer Prerequisites*.

6.10.3 Inference

Inferences express the contract between the improvers and the improve runner framework. An inference is intended to contain data points about a vulnerability without any uncertainties, which means that one inference will target one vulnerability with the specific relevant affected and fixed packages (in the form of PackageURLs). There is no notion of version ranges here: all package versions must be explicitly specified.

Because this concrete relationship is rarely available anywhere upstream, we have to *infer* these values, thus the name. As inferring something is not always perfect, an Inference also comes with a confidence score.

6.10.4 Improver

All the Improvers must inherit from `Improver` superclass and implement the `interesting_advisories` property and the `get_inferences` method.

6.11 Writing an improver

6.11.1 Locate the Source File

If the improver will be working on data imported by a specific importer, it will be located in the same file at `vulnerabilites/importers/importer-name.py`. Otherwise, if it is a generic improver, create a new file `vulnerabilites/improvers/improver-name.py`.

6.11.2 Explore Package Managers (Optional)

If your Improver depends on the discrete versions of a package, the package managers' VersionAPI located at `vulnerabilites/package_managers.py` could come in handy. You will need to instantiate the relevant `VersionAPI` in the improver's constructor and use it later in the implemented methods. See an already implemented improver (NginxBasicImprover) for an example usage.

6.11.3 Implement the `interesting_advisories` Property

This property is intended to return a QuerySet of `Advisory` on which the `Improver` is designed to work.

For example, if the improver is designed to work on Advisories imported by `ExampleImporter`, the property can be implemented as

```
class ExampleBasicImprover(Improver):

    @property
    def interesting_advisories(self) -> QuerySet:
        return Advisory.objects.filter(created_by=ExampleImporter.qualified_name)
```

6.11.4 Implement the `get_inferences` Method

The framework calls `get_inferences` method for every `AdvisoryData` that is obtained from the `Advisory` QuerySet returned by the `interesting_advisories` property.

It is expected to return an iterable of `Inference` objects for the given `AdvisoryData`. To avoid storing a lot of Inferences in memory, it is preferable to yield from this method.

A very simple Improver that processes all Advisories to create the minimal relationships that can be obtained by existing data can be found at `vulnerabilites/improvers/default.py`, which is an example of a generic improver. For a more sophisticated and targeted example, you can look at an already implemented improver (e.g., `vulnerabilites/importers/nginx.py`).

Improvers are not limited to improving discrete versions and may also improve `aliases`. One such example, improving the importer written in the *importer tutorial*, is shown below.

```python
from datetime import datetime
from datetime import timezone
from typing import Iterable

import requests
from django.db.models.query import QuerySet
from packageurl import PackageURL
from univers.version_range import NginxVersionRange
from univers.versions import SemverVersion

from vulnerabilities.importer import AdvisoryData
from vulnerabilities.improver import MAX_CONFIDENCE
from vulnerabilities.improver import Improver
from vulnerabilities.improver import Inference
from vulnerabilities.models import Advisory
from vulnerabilities.severity_systems import SCORING_SYSTEMS


class ExampleImporter(Importer):
    ...


class ExampleAliasImprover(Improver):
    @property
    def interesting_advisories(self) -> QuerySet:
        return Advisory.objects.filter(created_by=ExampleImporter.qualified_name)

    def get_inferences(self, advisory_data) -> Iterable[Inference]:
        for alias in advisory_data.aliases:
            new_aliases = fetch_additional_aliases(alias)
            aliases = new_aliases + [alias]
            yield Inference(aliases=aliases, confidence=MAX_CONFIDENCE)


def fetch_additional_aliases(alias):
    alias_map = {
        "CVE-2021-23017": ["PYSEC-1337", "CERTIN-1337"],
        "CVE-2021-1234": ["ANONSEC-1337", "CERTDES-1337"],
    }
    return alias_map.get(alias)
```

Note:

Use `make valid` to format your new code using black and isort automatically.

Use `make check` to check for formatting errrors.

6.11.5 Register the Improver

Finally, register your improver in the improver registry at `vulnerabilites/improvers/__init__.py`.

```
from vulnerabilities import importers
from vulnerabilities.improvers import default

IMPROVERS_REGISTRY = [
    default.DefaultImprover,
    importers.nginx.NginxBasicImprover,
    importers.example.ExampleAliasImprover,
]

IMPROVERS_REGISTRY = {x.qualified_name: x for x in IMPROVERS_REGISTRY}
```

Congratulations! You have written your first improver.

6.11.6 Run Your First Improver

If everything went well, you will see your improver in the list of available improvers.

```
$ ./manage.py improve --list

Vulnerability data can be processed by these available improvers:
vulnerabilities.improvers.default.DefaultImprover
vulnerabilities.importers.nginx.NginxBasicImprover
vulnerabilities.importers.example.ExampleAliasImprover
```

Before running the improver, make sure you have imported the data. An improver cannot improve if there is nothing imported.

```
$ ./manage.py import vulnerabilities.importers.example.ExampleImporter

Importing data using vulnerabilities.importers.example.ExampleImporter
Successfully imported data using vulnerabilities.importers.example.ExampleImporter
```

Now, run the improver.

```
$ ./manage.py improve vulnerabilities.importers.example.ExampleAliasImprover

Improving data using vulnerabilities.importers.example.ExampleAliasImprover
Successfully improved data using vulnerabilities.importers.example.ExampleAliasImprover
```

See *Command Line Interface* for command line usage instructions.

6.11.7 Enable Debug Logging (Optional)

For more visibility, turn on debug logs in `vulnerablecode/settings.py`.

```
DEBUG = True
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'root': {
        'handlers': ['console'],
        'level': 'DEBUG',
    },
}
```

Invoke the improve command now and you will see (in a fresh database, after importing):

```
$ ./manage.py improve vulnerabilities.importers.example.ExampleAliasImprover

Improving data using vulnerabilities.importers.example.ExampleAliasImprover
Running improver: vulnerabilities.importers.example.ExampleAliasImprover
Improving advisory id: 1
New alias for <Vulnerability: VULCOID-23dd9060-3bc0-4454-bfbd-d16c08a966a6>: PYSEC-1337
New alias for <Vulnerability: VULCOID-23dd9060-3bc0-4454-bfbd-d16c08a966a6>: CVE-2021-
↪23017
New alias for <Vulnerability: VULCOID-23dd9060-3bc0-4454-bfbd-d16c08a966a6>: CERTIN-1337
Improving advisory id: 2
New alias for <Vulnerability: VULCOID-fae4e06e-4815-45fe-ae95-8d2356ffb5b9>: CERTDES-1337
New alias for <Vulnerability: VULCOID-fae4e06e-4815-45fe-ae95-8d2356ffb5b9>: ANONSEC-1337
New alias for <Vulnerability: VULCOID-fae4e06e-4815-45fe-ae95-8d2356ffb5b9>: CVE-2021-
↪1234
Finished improving using vulnerabilities.importers.example.ExampleAliasImprover.
Successfully improved data using vulnerabilities.importers.example.ExampleAliasImprover
```

Note: Even though CVE-2021-23017 and CVE-2021-1234 are not supplied by this improver, the output above shows them because we left out running the `DefaultImprover` in the example. The `DefaultImprover` inserts minimal data found via the importers in the database (here, the above two CVEs). Run importer, DefaultImprover and then your improver in this sequence to avoid this anomaly.

CHAPTER

SEVEN

FAQ

7.1 During development, how do I quickly empty my database and start afresh ?

```
$ dropdb vulnerablecode
$ make postgres
```

CHAPTER

EIGHT

MISCELLANEOUS

8.1 Continuous periodic Data import

If you want to run the import periodically, you can use a systemd timer. Here is an example:

```
$ cat ~/.config/systemd/user/vulnerablecode.service

[Unit]
Description=Run VulnerableCode importers

[Service]
Type=oneshot
ExecStart=/path/to/venv/bin/python /path/to/vulnerablecode/manage.py import --all && /
↪path/to/venv/bin/python /path/to/vulnerablecode/manage.py improve --all

$ cat ~/.config/systemd/user/vulnerablecode.timer

[Unit]
Description=Periodically run VulnerableCode importers

[Timer]
OnCalendar=daily

[Install]
WantedBy=multi-user.target
```

Start this timer with:

```
systemctl --user daemon-reload
systemctl --user start vulnerablecode.timer
```

8.2 Environment variables configuration

VulnerableCode loads environment variables from an *.env* file when provided. VulnerableCode first checks the file at */etc/vulnerablecode/.env* and if not present, it will attempt to load a *.env* file from the checkout directory.

The file at */etc/vulnerablecode/.env* has precedence.

8.3 Throttling rate configuration

The default throttling settings are defined in `settings.py`.

To override the default settings, add env variables in `.env` file define the settings there. For example:

```
VULNERABLECODE_ALL_VULNERABLE_PACKAGES_THROTTLING_RATE = '1000/hour'
VULNERABLECODE_BULK_SEARCH_PACKAGE_THROTTLING_RATE = '10/minute'
VULNERABLECODE_PACKAGES_SEARCH_THROTTLING_RATE = '1000/second'
VULNERABLECODE_VULNERABILITIES_SEARCH_THROTTLING_RATE = '1000/hour'
VULNERABLECODE_ALIASES_SEARCH_THROTTLING_RATE = '1000/hour'
VULNERABLECODE_CPE_SEARCH_THROTTLING_RATE = '10/minute'
VULNERABLECODE_BULK_SEARCH_CPE_THROTTLING_RATE = '10/minute'
```

CHAPTER

NINE

IMPORTER OVERVIEW

Importers are responsible for scraping vulnerability data such as vulnerabilities and their fixes and for storing the scraped information in a structured fashion. The structured data created by the importer then provides input to an improver (see *Improver Overview*), which is responsible for creating a relational model for vulnerabilities, affected packages and fixed packages.

All importer implementation-related code is defined in `vulnerabilites/importer.py`.

In addition, the framework-related code for actually invoking and processing the importers is located in `vulnerabilites/import_runner.py`.

The importers, after scraping, provide `AdvisoryData` objects. These objects are then processed and inserted into the `Advisory` model.

While implementing an importer, it is important to make sure that the importer does not alter the upstream data at all. Its only job is to convert the data from a data source into structured – yet non-relational – data. This ensures that we always have a *true* copy of an advisory without any modifications.

Given that a lot of advisories publish version ranges of affected packages, it is necessary to store those ranges in a structured manner. *Vers* was designed to solve this problem. It has been implemented in the univers library whose development goes hand in hand with VulnerableCode.

The data imported by importers is not useful by itself: it must be processed into a relational model. The version ranges are required to be resolved into concrete ranges. These are achieved by `Improvers` (see *Improver Overview* for details).

As of now, the following importers have been implemented in VulnerableCode:

Importer Name	Data Source	Ecosystems Covered
rust	https://github.com/RustSec/advisory-db	rust crates
alpine	https://secdb.alpinelinux.org/	alpine packages
archlinux	https://security.archlinux.org/json	arch packages
debian	https://security-tracker.debian.org/tracker/data/json	debian packages
npm	https://github.com/nodejs/security-wg.git	npm packages
ruby	https://github.com/rubysec/ruby-advisory-db.git	ruby gems
ubuntu		ubuntu packages
retiredotnet	https://github.com/RetireNet/Packages.git	.NET packages
suse_backp	http://ftp.suse.com/pub/projects/security/yaml/	SUSE packages
debian_oval	https://www.debian.org/security/oval/	debian packages
redhat	https://access.redhat.com/hydra/rest/securitydata/cve.json	rpm packages
nvd	https://nvd.nist.gov/vuln/data-feeds#JSON_FEED	none
gentoo	https://anongit.gentoo.org/git/data/glsa.git	gentoo packages
openssl	https://www.openssl.org/news/vulnerabilities.xml	openssl
ubuntu_usn	https://usn.ubuntu.com/usn-db/database-all.json.bz2	ubuntu packages
github	https://api.github.com/graphql	maven, .NET, php-composer, pypi packages. ruby gems
msr2019	https://raw.githubusercontent.com/SAP/project-kb/master/MSR2019/dataset/vulas_db_msr2019_release.csv	maven packages
apache_httŗ	https://httpd.apache.org/security/json	apache-httpd
kaybee	https://github.com/SAP/project-kb.git	maven packages
nginx	http://nginx.org/en/security_advisories.html	nginx
postgresql	https://www.postgresql.org/support/security/	postgresql
elixir_secu	https://github.com/dependabot/elixir-security-advisories	hex packages
suse_scores	https://ftp.suse.com/pub/projects/security/yaml/suse-cvss-scores.yaml	vulnerability severity scores by SUSE
mozilla	https://github.com/mozilla/foundation-security-advisories	mozilla
mattermost	https://mattermost.com/security-updates/	mattermost server, desktop and mobile apps

CHAPTER

TEN

IMPROVER OVERVIEW

Improvers improve upon already imported data. They are responsible for creating a relational model for vulnerabilites and packages.

An Improver is intended to contain data points about a vulnerability and the relevant discrete affected and fixed packages (in the form of PackageURLs). There is no notion of version ranges here; all package versions must be explicitly specified. As this concrete relationship might not always be absolutely correct, improvers supply a confidence score and only the record with the highest confidence against a vulnerability and package relationship is stored in the database.

There are two categories of improvers:

- **Generic**: Improve upon some imported data irrespective of any importer. These improvers are defined in `vulnerabilites/improvers/`.
- **Importer Specific**: Improve upon data imported by a specific importer. These are defined in the corresponding importer file itself.

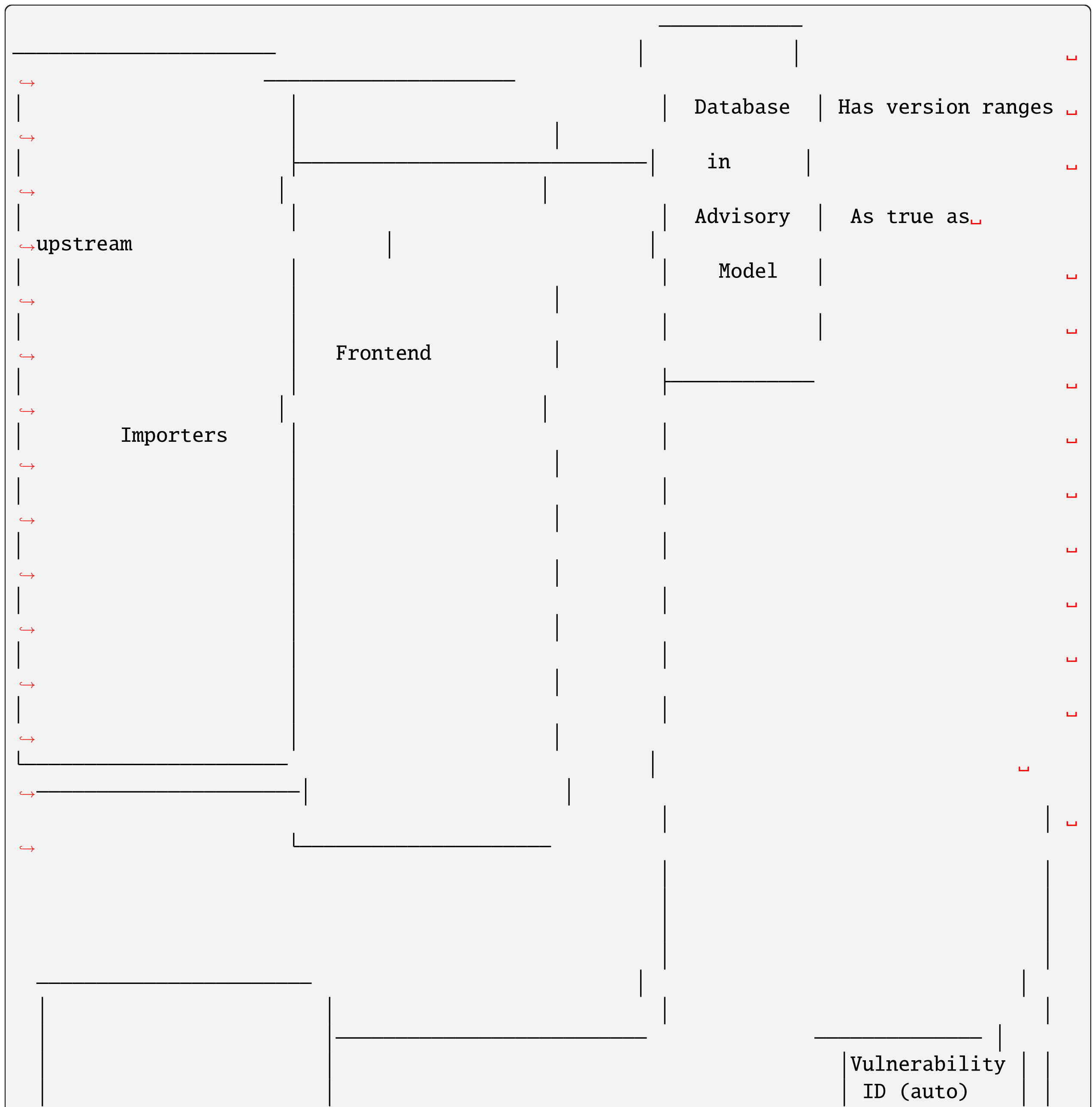
Both types of improvers internally work in a similar fashion. They indicate which `Advisory` they are interested in and when supplied with those Advisories, they return Inferences. An `Inference` is more explicit than an `Advisory` and is able to answer questions like "Is package A vulnerable to Vulnerability B ?". Of course, there is some confidence attached to the answer, which could also be `MAX_CONFIDENCE` in certain cases.

The possibilities with improvers are endless; they are not restricted to take one approach. Features like *Time Travel* and *finding fix commits* could be implemented as well.

You can find more in-code documentation about improvers in `vulnerabilites/improver.py` and the framework responsible for invoking these improvers in `vulnerabilites/improve_runner.py`.

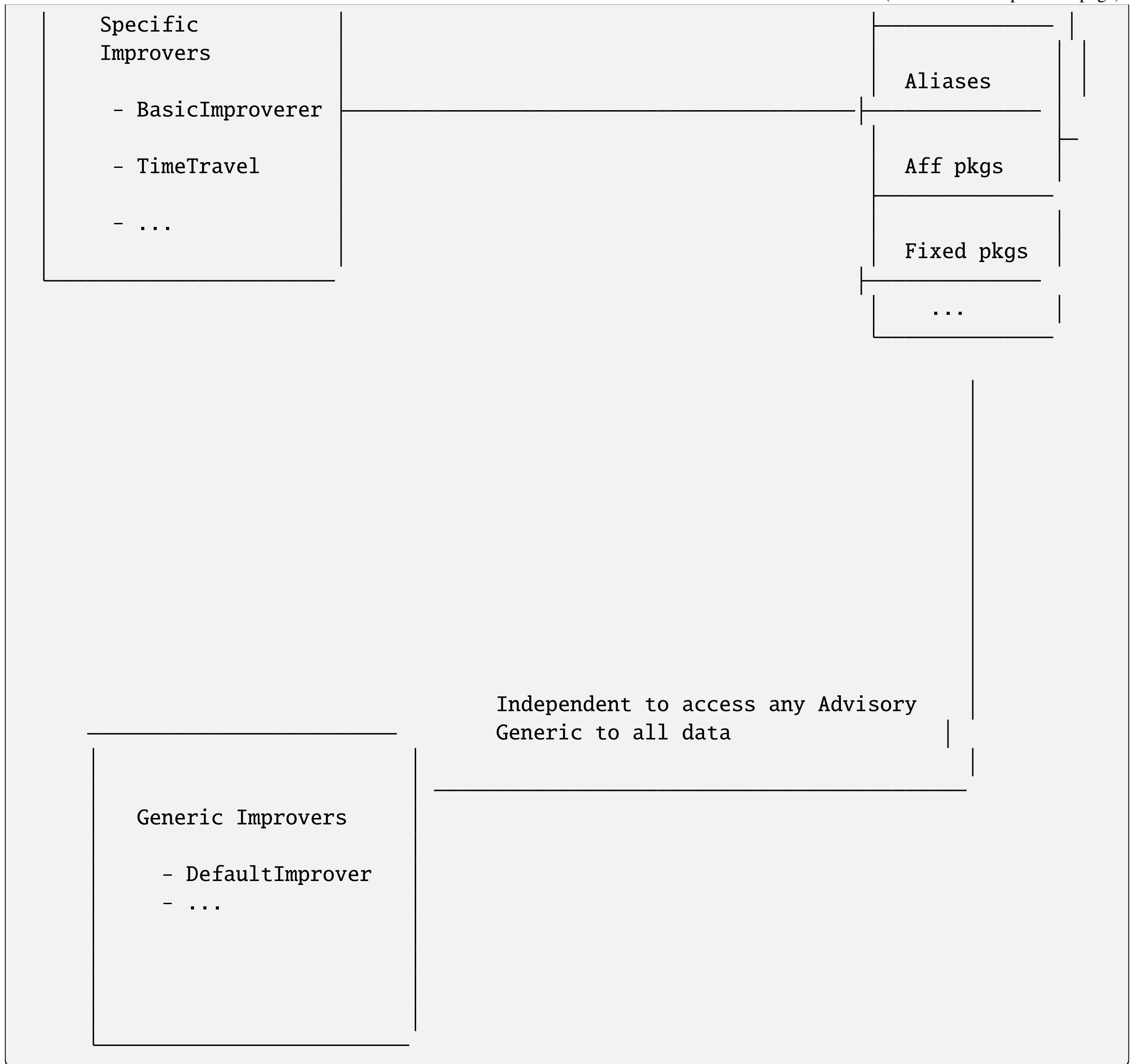
CHAPTER ELEVEN

FRAMEWORK OVERVIEW



(continues on next page)

(continued from previous page)



CHAPTER

TWELVE

COMMAND LINE INTERFACE

The main entry point is the Django *manage.py* management command script.

12.1 $ ./manage.py --help

Lists all sub-commands available, including Django built-in commands. VulnerableCode's own commands are listed under the [vulnerabilities] section:

```
$ ./manage.py --help
...
[vulnerabilities]
    import
    improve
    purl2cpe
```

12.2 $ ./manage.py <subcommand> --help

Displays help for the provided sub-command.

For example:

```
$ ./manage.py import --help
usage: manage.py import [-h] [--list] [--all] [--version] [-v {0,1,2,3}]
                        [--settings SETTINGS] [--pythonpath PYTHONPATH]
                        [--traceback] [--no-color] [--force-color]
                        [--skip-checks]
                        [sources [sources ...]]

Import vulnerability data

positional arguments:
  sources               Fully qualified importer name to run
```

12.3 $ ./manage.py import <importer-name>

Import vulnerability data using the given importer name.

Other variations:

- `--list` List all available importers
- `--all` Run all available importers

12.4 $ ./manage.py improve <improver-name>

Improve the imported vulnerability data using the given improver name.

Other variations:

- `--list` List all available improvers
- `--all` Run all available improvers

12.5 $ ./manage.py purl2cpe --destination <directory

Dump a mapping of CPEs to PURLs grouped by vulnerability in the `destination` directory.

Other variations:

- `--limit` Limit the number of processed vulnerabilities

CHAPTER

THIRTEEN

IMPORTERS

Importer Name	Data Source	Ecosystems Covered
rust	https://github.com/RustSec/advisory-db	rust crates
alpine	https://secdb.alpinelinux.org/	alpine packages
archlinux	https://security.archlinux.org/json	arch packages
debian	https://security-tracker.debian.org/tracker/data/json	debian packages
npm	https://github.com/nodejs/security-wg.git	npm packages
ruby	https://github.com/rubysec/ruby-advisory-db.git	ruby gems
ubuntu		ubuntu packages
retiredot-net	https://github.com/RetireNet/Packages.git	.NET packages
suse_backp	http://ftp.suse.com/pub/projects/security/yaml/	SUSE packages
de-bian_oval	https://www.debian.org/security/oval/	debian packages
redhat	https://access.redhat.com/hydra/rest/securitydata/cve.json	rpm packages
nvd	https://nvd.nist.gov/vuln/data-feeds#JSON_FEED	none
gentoo	https://anongit.gentoo.org/git/data/glsa.git	gentoo packages
openssl	https://www.openssl.org/news/vulnerabilities.xml	openssl
ubuntu_usn	https://usn.ubuntu.com/usn-db/database-all.json.bz2	ubuntu packages
github	https://api.github.com/graphql	maven, .NET, php-composer, pypi packages. ruby gems
msr2019	https://raw.githubusercontent.com/SAP/project-kb/master/MSR2019/dataset/vulas_db_msr2019_release.csv	maven packages
apache_http	https://httpd.apache.org/security/json	apache-httpd
kaybee	https://github.com/SAP/project-kb.git	maven packages
nginx	http://nginx.org/en/security_advisories.html	nginx
post-gresql	https://www.postgresql.org/support/security/	postgresql
elixir_secu	https://github.com/dependabot/elixir-security-advisories	hex packages
suse_scores	https://ftp.suse.com/pub/projects/security/yaml/suse-cvss-scores.yaml	vulnerability severity scores by SUSE
mozilla	https://github.com/mozilla/foundation-security-advisories	mozilla
matter-most	https://mattermost.com/security-updates/	mattermost server, desktop and mobile apps

CHAPTER

FOURTEEN

GOOGLE SUMMER OF CODE 2021 FINAL REPORT

14.1 Organization - AboutCode

Hritik Vijay
Project: VulnerableCode

14.2 Overview

VulnerableCode is a decentralized python program to collect data about open source software vulnerabilities across the internet. My proposal for this year's Google Summer of Code involved improving the import speed, refactoring existing code, finding points for overall improvement and adding importers.

14.3 Detailed Report

14.3.1 Improve Import Time

Profiling showed that a lot of time was being wasted during auto commits undertaken by django. Wraping the importer in an atomic block avoids lots of database commits and shows huge performance improvement. This simple change allows for much faster import times while not drastically changing the code structure:

```
Alpine: 202.7s -> 50.9s
Archlinux 2116.6s -> 107.8s
Gentoo 3176.3s -> 225.8s
```

Yielding an average of 93% reduction in time (14x faster)

More: https://github.com/nexB/vulnerablecode/pull/478

14.3.2 Speed up upstream tests

VulnerableCode performs upstream tests for all the importers to make sure that any change change in upstream data structure is easily spotted. This allows us to have a look at failing importers without actually deploying the application.

Earlier, all of the importers were run one by one in order to verify that they are intact. While this being the obvious and the full proof way to detect any anomalies in the imported data schema, it did not work because the time required to run all the importers much exceeded 6 hours - which is the maximum time allowed for GitHub actions to run. With this PR, the updated_advisories method of each importer is expected to create at least one Advisory object. If it does so, the importer is marked working. While this is not full proof, it stays much below the allowed resource usage cap. In the end, this is a trade off between resource usage and data accuracy. This brings major performance improvement during the test.

Before: ~6hrs, now ~9 minutes
More: https://github.com/nexB/vulnerablecode/pull/490

14.3.3 Improve Docker Configuration

The preferred mode of deployment for VulnerableCode is deploying using Docker images. Docker configuration existing earlier was very insecure and rudimentary. I took the inspiration for a uniform Docker configuration from the ScanCodeIO project and provided with detailed documentation for installation using a docker image. The current configuration makes use of files like `docker.env` to supply container's environment and `.dockerignore` to skip over any unnecessary files for deployment.

More:
https://github.com/nexB/vulnerablecode/pull/497
https://github.com/nexB/vulnerablecode/pull/521

14.3.4 Add Makefile

Makefile usage is prevalent in sister projects like ScanCodeIO. It gives VulnerableCode a consistent behavior and provides a very friendly interface for invocations. This also avoids security risks like having a default django `SECRET_KEY` as it can be easily generated by a make target. I added a Makefile which has a similar usage as that of ScanCodeIO, replaced all the CI tests to use make, updated the relevant part of the documentation and updated settings to reject insecure deployments.

More:
https://github.com/nexB/vulnerablecode/pull/497
https://github.com/nexB/vulnerablecode/pull/523

14.3.5 Use svn to collects tags in GitHubTagsAPI

Surprisingly, GitHub allows svn requests to repositories. Now we can have all the tags with a single request. This is much more efficient and gentle to the APIs. This was as issue since the importers based on GithubDataSource were failing because of being rate limited by GitHub.

Philippe, thank you so much for the suggestion
More: https://github.com/nexB/vulnerablecode/pull/508

14.3.6 Separate import and improve operations - WIP

This introduces a new concept of `improver`. Earlier, data fetching and improvement were done as one single process by `importer`. This meant that importers were convoluted and not very modular. The concept of `improver` comes from the idea that an `importer` should only do one thing - import. Any further improvement on the data is delegated to the improvers. This allows for us to have multiple ways of improvement with certain confidence on the improved data making the import and improve operations modular and simpler to work with. As a bonus, writing importers will be very easy and welcome more contributors to the project. As of writing this report, this remains a work in progress which will be finished very soon.

More: https://github.com/nexB/vulnerablecode/pull/525

14.3.7 Others

- helper: split_markdown_front_matter: https://github.com/nexB/vulnerablecode/pull/443
- Dump yaml in favor of saneyaml https://github.com/nexB/vulnerablecode/pull/452
- Refactor package_managers https://github.com/nexB/vulnerablecode/pull/495/commits
- Importers bugfix https://github.com/nexB/vulnerablecode/pull/505

14.4 Pre GSoC

I started to like VulnerableCode as soon as I laid eyes on the project. While exploring the codebase, I realized that there is a lot of room for improvement. Thus I looked for simple improvements and bugs to fix in the early stage, which were:

- Correct API docs path and fix pytest invocation
- Explicity provide lxml parser to beautifulsoup
- Make sure vulnerability id is_cve or is_vulcoid
- Fix istio importer (cleared a huge confusion about the codebase)
- Add me to AUTHORS (Should've done this a lot earlier)
- Add unspecified scoring system
- Fix redhat import failure (This one took a *lot* of effort to pinpoint)
- expose find_all_cve helper

14.5 Post GSoC - Future Plans and what's left

I wish to carry on with the development of VulnerableCode and implement the ideas suggested by my mentors. This will require a lot of effort to bring VulnerableCode to a stable point. I hope to see VulnerableCode integrated into the ScanCode toolkit happen in a near future.

Further, if possible, I would like VulnerableCode to interact with other great open source tools like *Eclipse Steady* and *Prospector*. VulnerableCode, currently, works statically to collect all the vulnerabilities from different data sources, meanwhile there have been some developments with the Prospector project of Eclipse Steady. The project aims to scan fix-commits of the git repository in order to find out if the vulnerable part of a library was actually used in a project. It is not always the case that if a library is vulnerable then all the projects building upon it would be vulnerable too. It is crucial to identify if it is worth updating the library in use and dealing with the breaking changes. *Prospectus* is undergoing improvements in order to be released as a usable public tool. *Project KB* (Under Eclipse Steady) is also working on a "tool support for mining repositories and databases of advisories to establish the (missing) link between vulnerabilities (as described in natural language in the advisories) and the corresponding fix-commits". When these projects are ready for public use I would like to add them to VulnerableCode as a modules. I hope this will benefit both the projects and the downstream.

After everything mentioned above, writing importers and improvers is something that is still left. In my opinion, this needs to be addressed after having a stable structure for VulnerableCode.

14.6 Closing Thoughts

I really enjoyed working on the project. There were ups and downs when I met some weird bugs but every one of them taught me something new about Python, Django and programming in general. The best part of working with my amazing mentors - Philippe and Shivam - were the weekly meets where we would together try to figure out how to proceed with the development. I learned something new with every call and interaction we had. Thank you so much my mentors for providing a very smooth experience and Google for showing me the guiding light for participation.

To the reader, I would really like you to read this before Philippe asks you to ;)

CHAPTER FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search