ptp Documentation

Release 0.1.1

Tao Sauvage

August 26, 2014

Contents

CHAPTER 1

Welcome!

Here we present the `ptp` (*Pentesters' Tools Parser*) project and answer the *What is it? What does it do? Why does it do it? How does it do it?* questions.

The project has been developed during the Google Summer of Code 2014, 10th edition, in order to create an automated ranking system for the OWASP - OWTF project.

1.1 OWASP - OWTF in a word

The OWASP - OWTF project provides an efficient approach to combine the power of automation with the out-of-the-box thinking that only the user can provide.

It gathers a complete set of plugins and merges their results into an interactive report. The user has then the possibility to add notes, to change details and to add media like screenshots in order to have a complete report.

1.2 The goals aimed by `ptp`

The primary goal of `ptp` is to enhance OWASP - OWTF in order to provide an automated ranking for each plugin. This will allow the user to focus attention on the most likely weak areas of a web application or network first, which will be valuable to efficiently use the remaining time in a penetration assessment.

Instead of evaluating every plugins run by OWASP - OWTF and defining the rankings for each of them, thanks to `ptp`, the user will be able to focus on the ones that have been ranked with the highest risks. The user is then able to confirm or override the automated rankings since we estimate that she/he is the only one that can accurately detect the false positives.

When developing the automated ranking system, `ptp`'s main goal was joined with a secondary one. Apart from its main feature which is **ranking the results from security tools reports**, it also provides an **unified way to reuse these reports directly in your python code**, without having to deal with complex parsing.

Note: The long-term objective for `ptp` is to support all security tools and tests. But `ptp` is in its early development phase and only supports the main ones for now.

CHAPTER 2

Installation

2.1 Using pip

The `ptp` library is available on PyPI at the following address: https://pypi.python.org/pypi/ptp.

The easiest way to install it is using pip.

```
$ pip install ptp
```

Note: If an error occurs during the installation process, check your permissions. It might be required to run pip as root.

2.2 From scratch

It is also possible to install the library from its repository. You will then be able to use the latest possible version or even try the develop branch.

The first step is to clone the repository of the project:

```
$ git clone https://github.com/owtf/ptp.git
```

Then run the *setup.py* script:

```
$ ./setup.py install
```

CHAPTER 3

Basic usage

3.1 Auto-detection mode

The `ptp` module provides the `ptp.PTP` class that exposes the public API of the library.

The simplest way to use `ptp.PTP` is with the **auto-detection mode**. This mode tries to reduce as much as possible our work by auto-detecting which tool has generated a given report and use the corresponding `libptp.parser.AbstractParser`.

That way, we do not need to know if the report we want to parse has been generated by W3AF, DirBuster or even Skipfish.

Example:

```
>>> from ptp import PTP
>>> myptp = PTP(pathname='my/directory', filename='my_report')
>>> myptp.parse()
[{'ranking': 4}, ..., {'ranking': 3}, ..., {'ranking': 1}]
```

Note: In the example above, the filename could have been omitted. In that case, `ptp` would have recursively walked into the directory *pathname* until a file would have matched one supported tool.

For instance, we could have done:

```
>>> from ptp import PTP
>>> myptp = PTP(pathname='my/directory')
>>> myptp.parse()
[{'ranking': 4}, ..., {'ranking': 3}, ..., {'ranking': 1}]
```

Be careful though, when omitting the *filename* parameter, `ptp` will stop as soon as a supported report file will be found! (i.e. `ptp` will not parse all the files in the *pathname* directory.)

If we are only looking for the highest risk that is listed in the report, we can use the following function:

```
>>> myptp.get_highest_ranking()
4
>>> from libptp.constants import HIGH
>>> myptp.get_highest_ranking() == HIGH
True
```

Note: To know the possible ranking values, please refer to the *Constants* section.

3.2 Explicit mode

If we already know which tool has generated the report, we can explicitly give that information to `ptp.PTP`. That will even speed up the whole process since it will not have to lookup for the right parser.

The list of the supported tools can be found like below:

```
>>> PTP.supported
{
    'arachni': [<class 'libptp.tools.arachni.parser.ArachniXMLParser'>],
    'dirbuster': [<class 'libptp.tools.dirbuster.parser.DirbusterParser'>],
    'metasploit': [<class 'libptp.tools.metasploit.parser.MetasploitParser'>],
    'nmap': [<class 'libptp.tools.nmap.parser.NmapXMLParser'>],
        <class 'libptp.tools.wapiti.parser.WapitiXMLParser'>,
        <class 'libptp.tools.wapiti.parser.Wapiti221XMLParser'>
    ],
    'owasp-cm-008': [<class 'libptp.tools.owasp.cm008.parser.OWASPCM008Parser'>],
    'robots': [<class 'libptp.tools.robots.parser.RobotsParser'>]
    'skipfish': [<class 'libptp.tools.skipfish.parser.SkipfishJSParser'>],
    'wapiti': [
    'w3af': [<class 'libptp.tools.w3af.parser.W3AFXMLParser'>],
}
```

Warning: The current support to Nmap does not provide any ranking yet. Refer to the *Nmap* section for more information.

Example:

```
>>> myptp = PTP('skipfish')
>>> myptp.parse(pathname='my/other/directory')
[{'ranking': 2}, {'ranking': 2}, {'ranking': 1}]
```

3.3 Attributes

If we are interested in the name of the tool that generated the report, it is stored in the `ptp.PTP.tool_name` attribute and can be retrieved like below:

```
>>> print(myptp.tool_name)
arachni  # In our case, it is Arachni that has generated our report.
```

We can also retrieve the list of the vulnerabilities thanks to the `ptp.PTP.vulns` attribute:

```
>>> myptp.vulns
[{'ranking': 4}, ..., {'ranking': 3}, ..., {'ranking': 1}]
```

And the metadata thanks to the `ptp.PTP.metadata` attribute.

```
>>> myptp.metadata
{'version': 'a.b'}
```

CHAPTER 4

Unit tests

The `ptp` module can be tested by running the *run_tests.py* python script.

```
$ ./run_tests.py
```

Note: Make sure the `./setup.py install` has been successful before running the script.

This script will run every existing unit tests that have been created for the module. If an error occurs, the string *FAIL* will be outputted in the terminal.

Note: It is possible to specify which unit test to run by specifying the name of the tool.

```
$ ./run_tests.py arachni
```

Using the command above only runs the unit tests for Arachni.

CHAPTER 5

Hack into ptp

5.1 Write our own support

Here we explain how we can contribute to the project by hacking into ptp's source code and enhance its list of supported tools.

First of all, we have to write a parser for our target tool. In our case, let us assume that the tool is named *MyTool* and that we want to parse its XML formatted reports.

The parser source code must be saved into the *tools/<tool name>/* and be named *parser.py*. Therefore, the parser for *MyTool* will be saved under the name *tools/mytool/parser.py*.

5.2 MyXMLParser class

In order for ptp to correctly retrieve the information that are contained in a tool report, it needs a specialized parser.

Let's start by writing the skeleton of our parser class. Since we are aiming to support *MyTool*'s XML reports, *XMLParser* seems to be the best class from which to inherit.

The *XMLParser* already defines libptp.parser.XMLParser.handle_file() for us. This will initialize the MyXMLParser.stream instance variable with a handle on the root node of the file.

5.2.1 The skeleton

By convention, the class name must contain the format it parses (in our case *XML*).

```
from libptp.parser import XMLParser


class MyXMLParser(XMLParser):
    """Specialized parser for MyTool."""

    __tool__ = 'mytool'
    __version__ = ['0.1']

    def __init__(self, pathname, filename='*.xml'):
        """Initialize MyXMLParser.

        :param str pathname: Path to the report directory.
        :param str filename: Regex matching the report file.
```

```
        """
        XMLParser.__init__(self, pathname, filename)
```

We added a couple of class attributes in order to give some information about what tool is parsed by our class and the supported versions.

Since our parser inherits from *XMLParser*, we do not have to specify the *__format__* class attribute, which is already set to *xml*.

Note: In order to keep the tool name homogene with the rest of the code base, *__tool__* must be lowercased.

Also, both the *__format__* and the *__version__* attributes are optional.

For instance *__version__* is optional because some tools don't provide such information (e.g. robots.txt).

5.2.2 Matching the supported reports

The next step is to write the `is_mine()` class method which tells `ptp` whether or not it can parse the report file.

Let us say that *MyTool*'s XML report has *<mytool version='x.x'>* as the root XML tag.

Therefore, our `is_mine()` function is:

```
class MyXMLParser(XMLParser):
    """Specialized parser for MyTool."""

    __tool__ = 'mytool'
    __version__ = ['0.1']

    # Omitted unchanged code

    @classmethod
    def is_mine(cls, pathname, filename='*.xml'):
        """Check if it is a supported MyTool report.

        :param str pathname: Path to the report directory.
        :param str filename: Regex matching the report file.

        :return: `True` if it supports the report, `False` otherwise.
        :rtype: :class:`bool`

        """
        try:
            stream = cls.handle_file(pathname, filename)
        except (ValueError, LxmlError):
            # If an error occurs when trying to open the file, then the
            # parser cannot deal with it.
            return False
        # The root tag must contain 'mytool'.
        if not cls.__tool__ in stream.tag:
            return False
        # Check if the root node has a 'version' attribute.
        if not 'version' in stream:
            return False
        # Check if the version is the one this parser supports.
        if not stream.get('version') in cls.__version__:
            return False
        return True
```

5.2.3 Parsing methods

Each *AbstractParser* class has to provide two methods:

- `libptp.parser.AbstractParser.parse_metadata()` which parses the metadata of the report and formats them into a `dict`.
- `libptp.parser.AbstractParser.parse_report()` which parses the discoveries that are listed in the report and formats them into a `list` of `dict`.

In order to keep it simple, we will not detail the implementations of these methods for our fake tool.

```
from libptp.parser import XMLParser


class MyXMLParser(XMLParser):
    """Specialized parser for MyTool."""

    __tool__ = 'mytool'
    __version__ = ['0.1']

    def __init__(self, pathname, filename='*.xml'):
        """Initialize MyXMLParser.

        :param str pathname: Path to the report directory.
        :param str filename: Regex matching the report file.

        """
        XMLParser.__init__(self, pathname, filename)

    @classmethod
    def is_mine(cls, pathname, filename='*.xml'):
        """Check if it is a supported MyTool report.

        :param str pathname: Path to the report directory.
        :param str filename: Regex matching the report file.

        :return: `True` if it supports the report, `False` otherwise.
        :rtype: :class:`bool`

        """
        try:
            stream = cls.handle_file(pathname, filename)
        except (ValueError, LxmlError):
            # If an error occurs when trying to open the file, then the
            # parser cannot deal with it.
            return False
        # The root tag must contain 'mytool'.
        if not cls.__tool__ in stream.tag:
            return False
        # Check if the root node has a 'version' attribute.
        if not 'version' in stream:
            return False
        # Check if the version is the one this parser supports.
        if not stream.get('version') in cls.__version__:
            return False
        return True

    def parse_metadata(self):
```

```
        return {}  # The expected behavior is to return a dict.

    def parse_report(self):
        return []  # The expected behavior is to return a list.
```

5.3 Tell ptp

Now that *MyTool* is supported thanks to our implementation of *MyXMLParser*, we only have one more thing to do in order to finish.

We need to update the `ptp.supported` list attribute by inserting our *MyXMLParser* inside like shown below:

```
# Omitted imports

from tools.mytool.parser import MyXMLParser

class PTP(object):

    # Omitted lines

    supported = {

        # Omitted supported tools.

        'w3af': [W3AFReport],

        # Omitted supported tools.

        'mytool': [MyXMLParser]}
```

We have done it! We have written our own support to the tool *MyTool* and integrated that into `ptp`!

Congratulations!

CHAPTER 6

Documentation

6.1 PTP

6.2 Basic Parsers

6.2.1 AbstractParser

6.2.2 XMLParser

6.2.3 FileParser

6.2.4 LineParser

6.3 Exceptions

6.4 Constants

6.5 Arachni

6.5.1 Parser

6.6 DirBuster

6.6.1 Parser

6.6.2 Signatures

6.7 Metasploit

Note: Since Metasploit does not force the users to follow a specific syntax when writing a module, `PTP` needs to know which `plugin` has generated the report in order to find the right signature.

6.7.1 Parser

6.7.2 Signatures

6.8 Nmap

Warning: The development of the Nmap ranking system has been postponed to after the GSoC. For now, the classes below only parse the XML reports generated by Nmap but do not rank the discoveries.

6.8.1 Parser

6.9 OWASP

6.9.1 CM-008

OWASP-CM-008 tests the HTTP methods of a website that are available.

Parser

Signatures

6.10 Robots.txt

6.10.1 Parser

6.10.2 Signatures

6.11 Skipfish

6.11.1 Parser

6.12 W3AF

6.12.1 Parser

6.13 Wapiti

6.13.1 Parser

6.13.2 Signatures

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*