

Capstone Final Report

Winter 2021

Instructor: Dr. Samir Tout



Group 4

Alex Peplinsi, Antonio Bally, Brandon Paul, Kyle Purchase

Introduction	2
Client	2
Problem Statement	3
Solution Value and Benefits	3
Overall Design	3
5.1 Entropy	6
5.2. Design	6
5.3. Instructions for Replicating the Environment.	7
Results	8
6.2. Steps to replicate experiment(s)	8
6.3. Our Results	11
Conclusion	13
Appendix A – Instructions...	14
References	18

1. Introduction

In the last few years, Python has become an increasing interest to companies with its broad capabilities and interactivity. Python has a massive following of programmers and security analysts behind it, making it one of the most used programming languages out there. With the popularity of Python comes the problem of being targeted by malicious intent. Since Python has been integrated into many companies' development environments, concerns arise that hackers may exploit vulnerabilities stemming from Python in the development supply chain. Protecting against these attacks is becoming more necessary and companies are looking for solutions to prevent these kinds of attacks.

2. Client

Our client specializes in secure architecture, web application development, penetration testing, and several other skills. Since Python is a common language used in application development, it is a significant potential target for attackers to exploit. However, Python, being an interpreted language compiled at run-time, is an uncommon choice for attackers and is not well researched. Our client is currently concerned about the malicious interactions with downloading Python modules. The `setup.py` file intended for preparing a system to download the module is usually not looked over, making it a prime target for hackers to exploit. Our client wishes us to develop some mitigation against this by statically checking this `setup.py` file for malicious code before integrating a library into their environment.

3. Problem Statement

Developers should install Python libraries without exposing themselves to potentially malicious code hidden in some setup.py installation files. Currently, developers must manually download and analyze setup.py files by hand to ensure a basic level of security. Due to the speed of development, it is unfeasible to analyze all imported libraries. This results in a significant security flaw that attackers can exploit to gain arbitrary code execution resulting in data leaks. To check for malicious Python applications using external libraries, a Python code analyzer was developed with the specific intent of looking for malicious code in setup.py files.

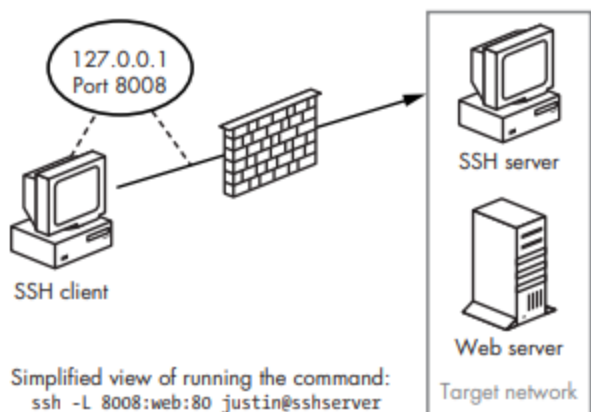
4. Solution Value and Benefits

Our solution provides the following benefits:

- a) Ability to quickly assess if a setup.py file contains potentially malicious code
- b) Provides a score to assess if there are malicious commands within the file
- c) Breaks down important elements of python programs into parsable json files
- d) Creates a readable HTML report detailing what was found in a python file

5. Overall Design

In trying to find malicious python scripts, we searched the Internet looking through coding forums, websites, and specific software resources such as Github. The greatest source of malicious code examples came from the Black Hat Python book written by Justin Seitz. This resource had 11 chapters and approximately 30 malicious python scripts that were used in helping define our baseline of questionable python code. Examples included python code that allows you to tunnel ssh traffic through another protocol (example illustrated below)..



```

def main():
    ❶ options, server, remote = parse_options()
    password = None
    if options.readpass:
        password = getpass.getpass('Enter SSH password: ')
    ❷ client = paramiko.SSHClient()
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy())
    verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
    try:
        client.connect(server[0], server[1], username=options.user, ~
            key_filename=options.keyfile, ~
            look_for_keys=options.look_for_keys, password=password)
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
        sys.exit(1)

    verbose('Now forwarding remote port %d to %s:%d ...' % (options.port, ~
        remote[0], remote[1]))

    ❸ try:
        reverse_forward_tunnel(options.port, remote[0], remote[1], ~
            client.get_transport())
    except KeyboardInterrupt:
        print('C-c: Port forwarding stopped.')
        sys.exit(0)

```

The book gave a description of the code and calls used. In addition, it did a deep dive into sections to help give us a clear understanding of how the code was being executed. The illustration below gives an example of a function review within the reverse SSH script.

The few lines at the top ❶ double-check to make sure all the necessary arguments are passed to the script before setting up the Parmakio SSH client connection ❷ (which should look very familiar). The final section in `main()` calls the `reverse_forward_tunnel` function ❸.

Let's have a look at that function.

```
def reverse_forward_tunnel(server_port, remote_host, remote_port, transport):
❶  transport.request_port_forward('', server_port)
    while True:
❷      chan = transport.accept(1000)
        if chan is None:
            continue
❸      thr = threading.Thread(target=handler, args=(chan, remote_host, ~
        remote_port))

        thr.setDaemon(True)
        thr.start()
```

In the case above, the script calls a reverse forward funnel function. It uses a “while the condition is true statement” to executive the next lines of python script to accept a transport using a channel from a remote host using a remote port. The information above and all the examples extracted from the Black Hat Python Book were tracked and recorded in a JSON data structure. All the information was recorded so that our program could read against it to compare code it is scanning compared to our repository of known malicious python code.

The syntax for running our experiment included a command and switches. The four types of filters that can be used with our program are:

- Analyze File with Default Filters
 - Example: `python main.py --target <path to target file>`
- Analyze File with Custom Filters
 - Example: `python main.py --target <path to target file> --filters <path to filters file>`
- Analyze File and Show Negative Matches in Report
 - Example: `python main.py --target <path to target file> --keepNegatives`
- Run a Test Analysis and Generate a Test Report
 - Example: `python main.py --test`

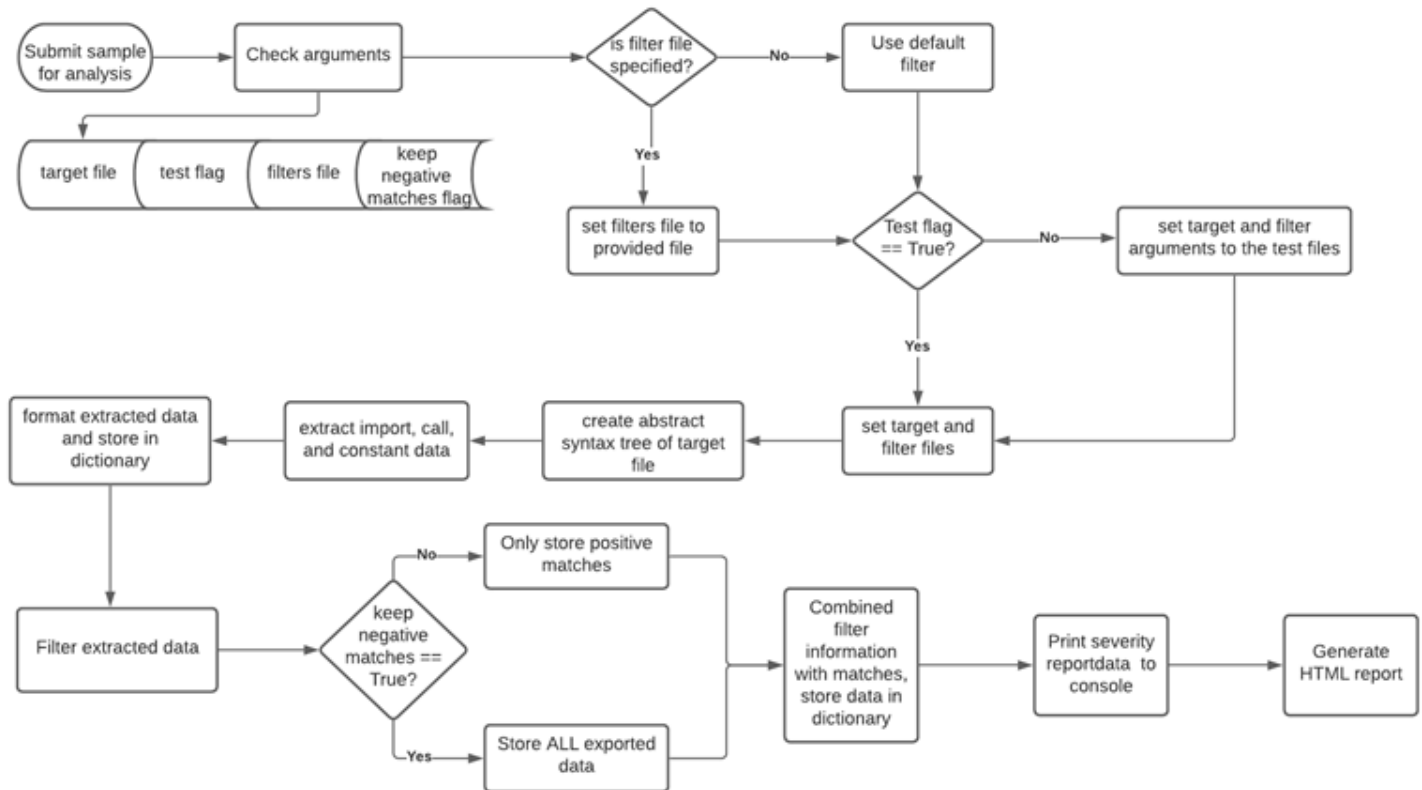
5.1 Entropy

The code calculates entropy of the setup.py file or any file using Shannon Entropy. Shannon's Entropy quantifies the amount of information in a variable. It provides a baseline for a theory around the notion of information. To do this, Pyteria reads the whole file into a byte array, then calculates the frequency of each byte value in the file and uses Shannon's entropy; which is $ent + freq * \log(freq, 2)$. We calculated the entropy of 300 known good .py files using the `ent` command built in Linux. We used the command to check the entropy of all 300 good .py files and export that information into a raw text file. The syntax of the command is "for i in *.py; do `ent -t "$i" >> out.txt;`". The output of this information will give us a baseline of what the normal entropy of a .py file should look like when scanning setup.py files with our application. The median entropy of the analyzed .py files was 4.676303, and we decided the program will flag any file above $Median + (Standard\ Deviation * 2)$ which will be any value greater than 5.296014741. The raw data and statistics can be found in *Pyteria/Data/etc/Entropy_Python_Data.elsx*.

5.2. Design

The diagram below details our approach to extracting code data from Python files. The program first checks for several arguments the user can specify which will modify what kind of output the user will receive. It will then use the built in `ast` library to create an abstract syntax tree (AST) of the python file. The AST creates a 'node' for each command issued in the file that can be visited using the `ast` library. The program pulls data from AST nodes we specify (currently import statements, method calls, and constants) and checks the data against filters we created to assign the file a severity score. This severity score is intended to be an easy to understand value which will help determine if the file may contain malicious code and should be investigated further. If

further analysis must be conducted, the analyzed data is stored in a json file. This file can easily be referenced; reducing the need to dig into source code.



5.3. Instructions for Replicating the Environment.

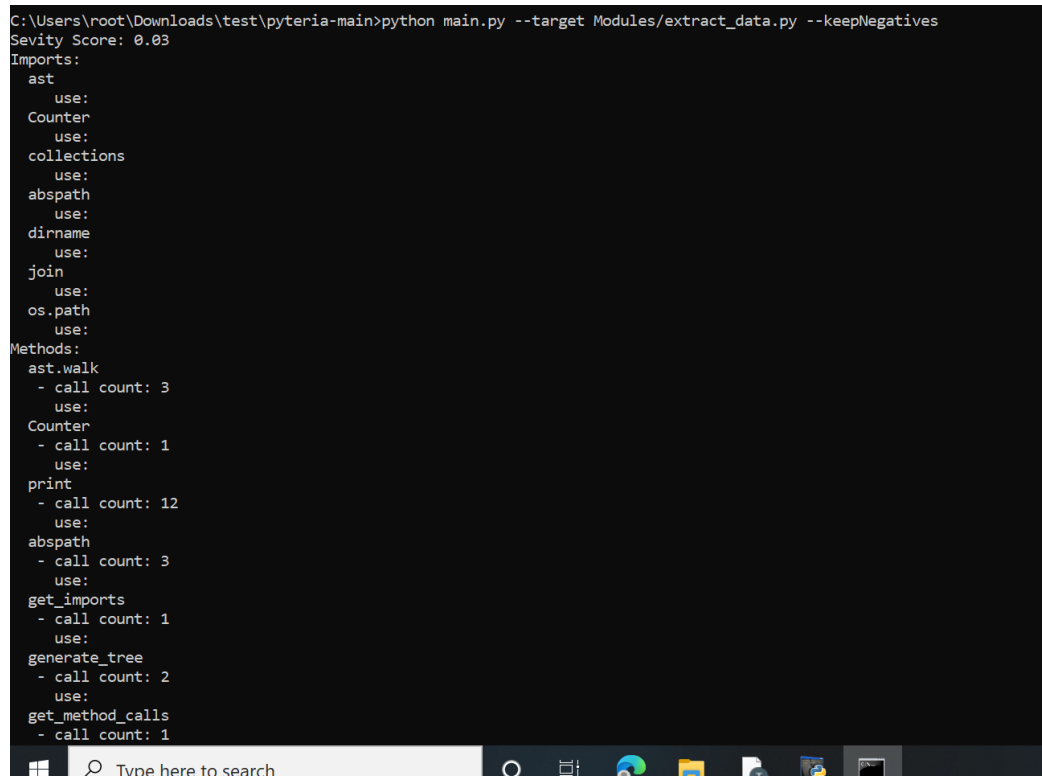
To replicate the environment, simply extract the provided zip folder into a dedicated directory and use the commands specified in section 6.2 to run the program. The HTML report along with the generated json files containing extracted data can be found in the '`<root>/Modules/Reports/`' directory.

6. Results

We utilized the environment above to conduct some testing for our proof of concept. For example, we ran our code against a sample python script file called `extract_sample.py`. The syntax for launching the scan is:

```
python main.py --target Modules/extract_data.py --keepNegatives
```

The output of this command generates a report with the severity rating of the code scanned, a list of all the components of the script (libraries, methods, calls, etc) examined, the entropy of the file, and a brief description of this examined piece of code and its use cases of how it's been used in the past.



```
C:\Users\root\Downloads\test\pyteria-main>python main.py --target Modules/extract_data.py --keepNegatives
Severity Score: 0.03
Imports:
  ast
    use:
  Counter
    use:
  collections
    use:
  abspath
    use:
  dirname
    use:
  join
    use:
  os.path
    use:
Methods:
  ast.walk
    - call count: 3
    use:
  Counter
    - call count: 1
    use:
  print
    - call count: 12
    use:
  abspath
    - call count: 3
    use:
  get_imports
    - call count: 1
    use:
  generate_tree
    - call count: 2
    use:
  get_method_calls
    - call count: 1
```

6.2. Steps to replicate experiment(s)

The steps below explain the various arguments that can be used, the commands required to analyze a file, run test output, and specify a filters file. To run any command, open a command line session and navigate to the root Pyteria directory.

Arguments

Flag	Purpose
--target	Specifies target file to run analysis on.
--filters	Specifies filter file to run target against.
--keepNegatives	Retain ALL extracted data for use in report. Use this to get a breakdown of what the target contains.
--test	Run a test analysis and generate a report using a test file and test filters.
--outDir	Directory to store generated report in.
--outName	Path to store generated report in.

<code>--jsonDir</code>	Directory to store generated json file.
<code>--jsonName</code>	Name for generated json file.
<code>--cssFile</code>	Path to CSS file to pair with html report.

Analyze File with Default Filters

```
python main.py --target <path to target file>
```

Analyze File with Custom Filters

```
python main.py --target <path to target file> --filters <path to filters file>
```

Analyze File and Show Negative Matches in Report

```
python main.py --target <path to target file> --keepNegatives
```

Run a Test Analysis and Generate a Test Report

```
python main.py --test
```

6.3. Our Results

Another part of our test examined methods and constants. The program would identify and rank the methods and constants scanned. In our example, we ran the command:

```
C:\Users\root\Downloads\test\pyteria-main>python main.py --test
```

And the following output was given:

```
C:\Users\root\Downloads\test\pyteria-main>python main.py --test
Sevity Score: 1.43
Imports:
  math
    use: When will I ever use this?
  re
    use: I'm pickle Reece
Methods:
  __import__
    - call count: 1
      use: Wierd way to import that, bro.
  random.randint
    - call count: 1
      use:
Constants:
  math
    use: When will I ever use this?
  1
    use: Some number
  gimme a number
    use: Just a string
C:\Users\root\Downloads\test\pyteria-main>
```

Here the number of imports and methods used, a brief description of their purpose, and a use case are displayed. The example illustrated is using beta code written for this proof of concept.

Through our research and experiment, we concluded a few finds. Frist, we had some issues and errors in our code that are illustrated below. Also, formatting of the report is outputted.

Severity <input type="text"/>	Level	Description
None	0	Doing nothing inherently malicious. i.e. print()
Low	1	Not being used maliciously, but facilitates other malicious activity/could be used maliciously.
Medium	2	Being used maliciously, but legitimate applications may need this as well.
High	3	Almost exclusively maliciously.
Critical	4	Always BAD.

Furthermore, we went through each library, method, or call and ranked it based on its use:

Chapter 2: The Network Basics

bhnet.py

Library Name	Severity	Use In program
getopt	1	Provides C style runtime arguments to Python. This is probably super strange to see in a setup.py file. Might be larger issues upon review.
socket	4	Connects client and host. Allows data to be sent over the network.

7. Conclusion

We have determined it is feasible to extract, analyze, and filter data from Python files for the purpose of identifying malicious commands. The most challenging tasks for this kind of analysis is 1. Finding malicious python code samples and 2. Maintaining a filters list that contains malicious commands, a severity value, and notes regarding how commands are generally used in malicious contexts. Similar to how an AntiVirus works with a signature file, These challenges may prove to be too resource intensive to justify and are generally less dynamic than other approaches. Given more time, approaches that utilize CVE databases, community reporting, or machine learning would likely yield more accurate results with less resources needed to generate and could potentially eliminate the manual intervention/review process required now. With all that said, we believe Pyteria is an excellent start to a static analysis framework for Python code. Malware analysis of setup.py files is only one part of what this program could potentially do if given more time. Having a framework to extract and format data from AST nodes opens the door to perform other security functions such as analyzing any python file for malicious commands or ensuring python programs are following best security practices without the need for dynamic analysis.

Appendix A – Instructions...












Example of Program Being Run

```
PS E:\GitHub\pyteria> python main.py --help
usage: main.py [-h] [--target TARGET] [--filters FILTERS] [--outDir OUTDIR] [--outName OUTNAME]





optional arguments:
  -h, --help            show this help message and exit
  --target TARGET        Target file to run analysis on.
  --filters FILTERS      Filter file to run target against.
  --outDir OUTDIR        Directory to store generated report in.
  --outName OUTNAME      Path to store generated report in.
  --jsonDir JSONDIR      Directory to store generated json file.
  --jsonName JSONNAME    Name for generated json file.
  --keepNegatives        Retains extracted data when compared against filters for use in report.
  --cssFile CSSFILE      Path to CSS file to pair with html report.
  --test                Test report generation using a test file and test filters.




PS E:\GitHub\pyteria> python main.py --test
Entropy Value: 4.95
Sevity Score: 4.00
Imports:
  math
    Note: When will I ever use this?
  re
    Note: I'm pickle Reece
Methods:
  __import__
    - call count: 1
      Note: Wierd way to import that, bro.
  random.randint
    - call count: 1
Constants:
PS E:\GitHub\pyteria> █
```

Example Contents of Json Data Directory

Applications 500GB SSD (E:) > GitHub > pyteria > Generated > Json Data				
Name	Date modified	Type	Size	
 extract_data.json	4/15/2021 00:40	JSON Source File	15 KB	
 extract_data-1.json	4/15/2021 00:41	JSON Source File	15 KB	
 extract_data-2.json	4/15/2021 00:50	JSON Source File	15 KB	
 test_cases.json	4/15/2021 00:47	JSON Source File	4 KB	
 test_cases-1.json	4/15/2021 00:48	JSON Source File	4 KB	
 test_cases-2.json	4/15/2021 00:49	JSON Source File	4 KB	
 test_cases-3.json	4/15/2021 00:50	JSON Source File	4 KB	
 test_cases-4.json	4/15/2021 13:42	JSON Source File	4 KB	
 test_cases-5.json	4/15/2021 13:43	JSON Source File	4 KB	
 test_cases-6.json	4/15/2021 13:43	JSON Source File	4 KB	
 test_cases-7.json	4/15/2021 13:43	JSON Source File	4 KB	

Example Contents of Reports Directory

Applications 500GB SSD (E:) > GitHub > pyteria > Generated > Reports				
Name	Date modified	Type	Size	
 extract_data	4/14/2021 23:46	File folder		
 Generate Report Test	4/14/2021 23:56	File folder		
 generate_report	4/14/2021 23:39	File folder		
 Test-Report	4/14/2021 23:05	File folder		

Applications 500GB SSD (E:) > GitHub > pyteria > Generated > Reports > extract_data				
Name	Date modified	Type	Size	
 extract_data_report.html	4/15/2021 00:50	Firefox HTML Doc...	19 KB	
 main.css	4/15/2021 00:50	Cascading Style S...	3 KB	
 pyteria-logo.png	4/15/2021 00:50	PNG File	2,523 KB	

Example report.html File

Severity score: 3.00

Entropy Value: 4.56

Imports:

ast

hashlib

Counter

collections

abspath

dirname

join

os.path

Methods:

ast.walk

- call count: 3

Counter

- call count: 1

Constants:

Name: generate_tree **Purpose:** generates a code object ast module can parse **Arguments:** file_path, file path **Returns:** code object

SHA256: a5b92e85361f19916ea42c09d21af322d86a0f8599c79b622d9aedd614daa2f2

Name: get_imports **Purpose:** Get import data from an ast node **Arguments:** tree, ast node **Returns:** list of ImportStatement and ImportFromStatement objects

SHA256: a30c015a19456563cd426fdd0e1301e169662eaaa3b5boaef41a5b930c53c8b3

Name: format_imports **Purpose:** Formats import object list into json style dictionary **Arguments:** import_obj_list, ImportStatement **Returns:** dict

SHA256: 6555c6bd6212ce3bb412ed0741bed1a14f3304391bd1a2d447a0b4c7541b9c8f

References

- Chilton, A. (2015) *Getting the Alexa top 1 million sites directly from the server, unzipping it, parsing the csv and getting each line as an array*: retrieved from:
gist.github.com/chilts/7229605 on 2/23/202
- Downing, Ben (2021). *Using Entropy in Threat Hunting: a Mathematical Search for the Unknown*: retrieved from redcanary.com/blog/threat-hunting-entropy/ on 2/22/2021
- FB36, (2021). *Shannon Entropy Calculation (Python recipe)* retrieved from:
code.activestate.com/recipes/577476-shannon-entropy-calculation/#c3
- Guruprasad, S. (2016) *Parsing Python Abstract Syntax Trees* retrieved from:
suhas.org/function-call-ast-python/ on 4/16/2021
- Haney, A (2021). *python-morgue* retrieved from:
github.com/adamhaney/python-morgue/blob/master/morgue.py on 4/16/2021
- Hartman, K., (2020) *calculate file entropy* retrieved from:
kennethghartman.com/calculate-file-entropy/ on 4/17/2021
- Journaldev.com (2021). *Python AST - Abstract Syntax Tree* retrieved from:
journaldev.com/19243/python-ast-abstract-syntax-tree on 4/16/2021
- Layman, M. (2021) *Deciphering Python: How to use Abstract Syntax Trees (AST) to understand code* retrieved from: mattlayman.com/blog/2018/decipher-python-ast on 4/16/2021

Seitz, J. (2015). *Black Hat Python: Python Programming for Hackers and Pentesters*, San

Francisco: No Starch Press, Inc.

Wright, J. (2021) *Hunting for Malicious Packages on PyPI* retrieved from:

jordan-wright.com/blog/post/2020-11-12-hunting-for-malicious-packages-on-pypi/