

Lazarus Group targets Aerospace and Defense with new Comebacker variant

enki.co.kr/en/media-center/blog/lazarus-group-targets-aerospace-and-defense-with-new-comebacker-variant



Executive Summary

- ENKI identified a new variant of Comebacker, initially identified following public reporting of a malicious domain.
- The malware is delivered via lure documents themed around prominent aerospace and defense organizations, indicating a targeted espionage campaign against this sector.
- Pivoting from the initial C&C infrastructure, we uncovered an additional C&C domain and a related Comebacker sample, suggesting the campaign has been active since at least March 2025.

1. Overview

In 2025-06, ENKI initiated an investigation based on [ThreatBookLabs](#)' reporting of a malicious domain, [office-theme\[.\]com](#), attributed to Lazarus Group. Analysis of .docx files hosted on this domain revealed a multi-stage malware infection chain deploying a new variant of the Comebacker backdoor.

By pivoting on the malware's C&C infrastructure, we identified an additional C&C domain and a related Comebacker sample that suggests the campaign has been active since at least March 2025.

This report provides an analysis of this new Comebacker variant, details the associated infrastructure, and tracks the malware's evolution over time.

1.1. Comebacker

Comebacker was first reported by [Google Threat Analysis Group](#) in a 2021 report on a campaign targeting security researchers. Functioning as a downloader and backdoor, it is designed to retrieve and execute DLLs payloads from a C&C server. Microsoft later [named the malware "Comebacker"](#), and it has since been attributed to Lazarus group.

Since its initial discovery in 2021, Comebacker has been observed in multiple campaigns. In 2024, variants were found embedded in malicious PyPI packages, demonstrating the threat actor's continued activity.

2. Malware Analysis

2.1. C&C Server Open Directory

We identified staging activity on the open directory at an open directory on [office-theme\[.\]com](#). This document initiates a multi-stage execution flow, ultimately leading to the in-memory execution of the final COMEBACKER payload. The full infection chain is detailed in the following subsections.


 Open directory listing on office-theme[.]com

Open directory listing on office-theme[.]com

While multiple files were present in the directory, only four files with **.bin** extensions were downloadable at the time of analysis. These files were identified as Microsoft Word documents, each containing a malicious VBA macro. Although the lure content varied, all four droppers deploy the same malicious payload.

2.2. Comebacker Dropper

When a victim opens one of the malicious `.docx` files and enables macros, the embedded VBA code executes. We extracted this macro code for analysis using the `olevba` tool.

 VBA macro code extracted via olevba



VBA macro code extracted via olevba


The macro decrypts and deploys two embedded components that are stored as large hexadecimal strings: a loader DLL and a decoy document. The decryption process involves a custom algorithm using XOR and bit-swapping operations. A Python script to replicate this decryption is available in **Appendix C, under "Comebacker Dropper Decryption Script"**.

The decrypted files are written to the following paths on the victim system:


- Loader: `C:\ProgramData\WPSOffice\wpsoffice_aam.ocx`
- Decoy document: `C:\ProgramData\Document\EDGE_Group_Interview_NDA.docx`

The macro then executes the loader by calling the ``LoadLibraryA`` API function and opens the decoy document.

We identified four distinct decoy documents leveraging themes related to the aerospace and defense sectors, including lures impersonating Edge Group, Indian Institute of Technology Kanpur (IIT Kanpur), and Airbus. This specific targeting strongly indicates the campaign's objective is espionage.

 Decoy document impersonating IIT Kanpur:
Guest_Lecture_Invitation_Format_IITK.docx

Decoy document impersonating IIT Kanpur: Guest_Lecture_Invitation_Format_IITK.docx

 Decoy document related to Airbus:
Airbus_C295_Integration_Document_for_TASL.docx

Decoy document related to Airbus: Airbus_C295_Integration_Document_for_TASL.docx

2.3. Comebacker Stage 1 Loader – wpsoffice_aam.cox

The `wpsoffice_aam.cox` file is the second-stage loader, employed to decrypt, persist, and execute the third stage of the infection chain.

The loader first decrypts an embedded payload using the ChaCha20 stream cipher. The static configuration for this decryption is as follows:

- key: ad9c5aca9977d04c73be579199a827049b6dd9840091ffe8e23acc05e1d4a657
- iv: edc9ce049daeba35b8687740
- counter: 1

A Python script to replicate this decryption is available in Appendix C, under "Comebacker Stage 1 Loader Decryption Script".

Following decryption, the loader decompresses the resulting data using the zlib library. The final payload is written to `C:\ProgramData\US0Shared\US0Private.dll`.



ChaCha20 decryption and writing of USOPrivate.dll

To establish persistence, the loader creates a shortcut (.lnk) to **USOPrivate.dll** in the user's Startup folder.

```
cmd.exe /C powershell -Command "$s = (New-Object -  
COMWScript.Shell).CreateShortcut('C:\\ProgramData\\USOShared\\Micro.lnk'); $s.TargetPath =  
'C:\\Windows\\System32\\rundll32.exe'; $s.Arguments = '\"[USOPrivate.dll path]\" LoadMimi  
\"C:\\Windows\\System32\\cmd.exe\"'; $s.Save()"
```

After creating the shortcut, the loader calls the LoadMimi function in USOPrivate.dll using rundll32.exe.

2.4. Comebacker Stage 2 Loader – USOPrivate.dll

USOPrivate.dll is the final loader in the infection chain. It decrypts the embedded Comebacker and executes it directly from memory.

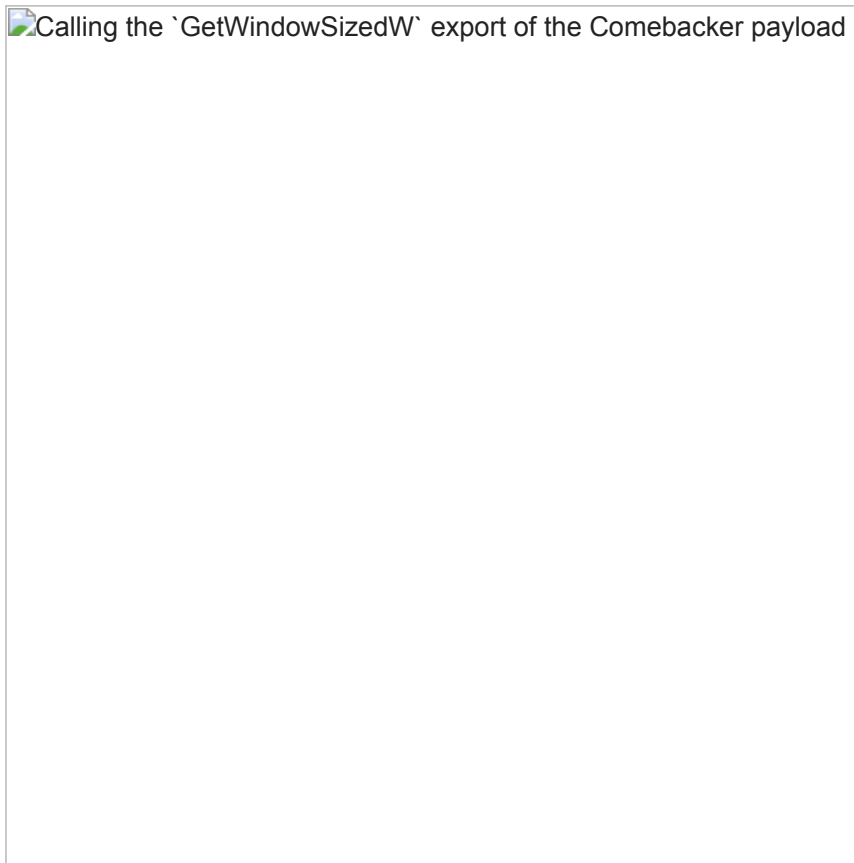
The DLL employs the same ChaCha20 decryption code seen in the previous stage, reusing the identical key, iv, and initial counter values.

- key: ad9c5aca9977d04c73be579199a827049b6dd9840091ffe8e23acc05e1d4a657
- iv: edc9ce049daeba35b8687740
- initial counter value: 1



Reuse of the ChaCha20 decryption code in USOPrivate.dll

After decryption and decompression, the loader loads the final Comebacker payload into memory. It then transfers execution to the payload by calling its `GetWindowSizeW` export with the string argument "1282".

Calling the `GetWindowSizedW` export of the Comebacker payload

Calling the GetWindowSizedW export of the Comebacker payload

2.5. Comebacker

Once executed, the Comebacker payload's main `GetWindowSizedW` export begins by generating a unique victim identifier. This ID is constructed by concatenating a randomly generated 10-character alphanumeric string, the argument value passed during execution ("`1282`"), and the static string "`64`".



Victim ID generation

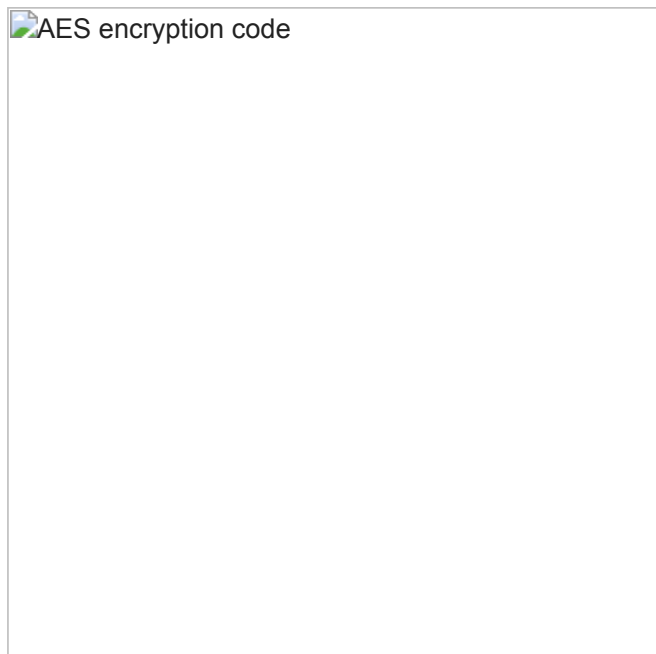
The malware then prepares to beacon out to its hardcoded C&C server:

```
hxxps://hiremployee[.]com
```

2.5.1. C&C Server Communication

All C&C communications occur over HTTPS. The outbound data is first encrypted with AES-128-CBC and then Base64-encoded. The malware uses the same value for both the encryption key and the IV.

encryption key and IV: x!P<&}mjH2YHRQ',



AES encryption code

Data received from the C&C server is similarly Base64-decoded and then decrypted using the same AES-128-CBC key and IV.

A Python script to decrypt this C&C traffic is available in Appendix C, under “**Comebacker C&C Data Decryption Script**”.

2.5.2. Initial Connection

The malware's initial beacon is encoded in the query string of the URL. The query string structure is as follows:

[random 2 lowercase letters]=[random 10 lowercase letters]&

[random 5 lowercase letters]=[base64-encoded ID value]&

[random 4 lowercase letters]=&

[random 6 lowercase letters]=0&

[random 6 lowercase letters]=[base64-encoded length of the current time]&

[random 6 lowercase letters]=[base64-encoded current time]&

[random letters up to 10]=[random letters up to 20]

The C&C server's response follows the following format.

[4 hexadecimal digits] [1 hexadecimal digit] [base64-encoded message length] [base64-encoded message]

The malware parses the hex digits and the decoded message to determine its next action. The primary behaviors are outlined below.

C&C server response	action
------------------------	--------

First value is 13	Terminates process.
Decoded message is "0"	Enter a sleep-retry loop. Beacons again after 60 seconds. After 20 consecutive "0" responses, the sleep interval increases to 20 minutes.
Decoded message is "1"	Beacon again after a short sleep of 10 seconds

C&C response info

If the server's response does not match any of the control commands, the malware downloads and executes a payload from the message with the following structure:

[command code][encrypted file size][export name][argument][MD5 hash of the encrypted file]

If a message that satisfies the above condition is received, it downloads the encrypted file from the C&C server and executes it.

2.5.3. File Download and Execution


Upon receiving a download and execute command, the malware requests the payload from the C&C server. After downloading, it first calculates the MD5 hash of the received encrypted file and compares it against the hash from the command. If the hashes do not match, the download is considered corrupt, and the malware re-enters its sleep-retry loop.

MD5 hash comparison

MD5 hash comparison

If the hashes match, the malware decrypts the payload using the same ChaCha20 implementation seen in the loader stages, with identical static key, nonce, and counter values.

- key: ad9c5aca9977d04c73be579199a827049b6dd9840091ffe8e23acc05e1d4a657
- iv: edc9ce049daeba35b8687740
- initial counter value: 1

 ChaCha20 decryption code

ChaCha20 decryption code

Finally, the decrypted payload is loaded into memory. The malware calls the exported function specified in the command, passing the provided argument. After execution completes, it sends the result back to the C&C server and resumes beaconing.

During our analysis, the C&C server did not respond with download and execute command, so we were unable to retrieve or analyze any next-stage payloads.

3. Additional Malware Collection and Analysis

To expand our visibility into the threat actor's infrastructure, we pivoted on known indicators. Using VirusTotal's Relations feature, we searched for other domains serving identical HTTP responses to the C&C server [hiremployee\[.\]com](#). This analysis identified a second C&C domain: [birancearea\[.\]com](#).

 VirusTotal Relations tab showing infrastructure overlap between C&C domains

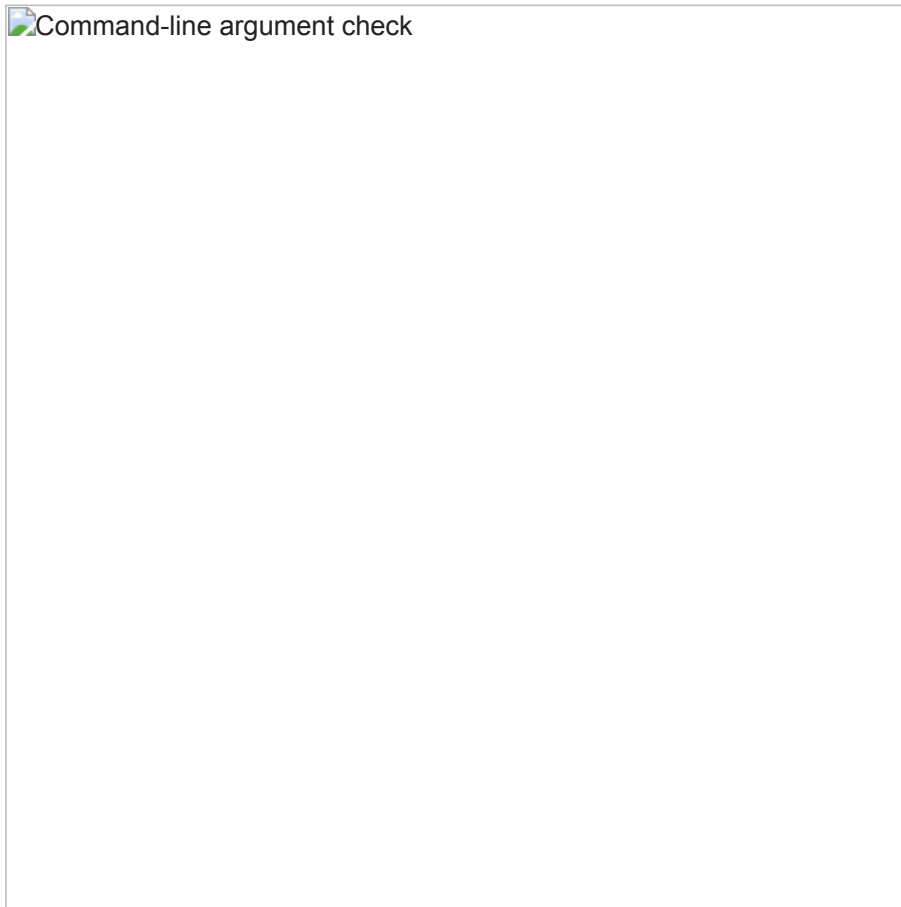
VirusTotal Relations tab showing infrastructure overlap between C&C domains

[birancearea\[.\]com](#) was scanned by VirusTotal in March 2025. We found an associated Comebacker sample that communicates with this domain, with the following hash.

3.1. Comebacker Stage 1 Loader

This loader is a DLL file that functions as the first stage in this alternate infection chain. It was first uploaded to VirusTotal in March 2025, and is similar to the Comebacker loader that was embedded in PyPI packages and distributed in 2024, including the HC256 implementation and usage, as well as the code that executes the decrypted payload.

Upon execution, the loader checks if the command line includes the specific argument `"9Ez6THDirL6Zye4"`. If this argument is not present, the process terminates. This check indicates the loader is designed to be executed by a preceding dropper or script, which we were unable to obtain.



Command-line argument check


If the argument is present, the loader decrypts an embedded payload using the HC256 stream cipher.

The decryption algorithm is HC256, and the hardcoded key/IV are identical to those used in the Comebacker loaders distributed via malicious PyPI packages in 2024.

key, iv: LH*x239udC<*sd_Sej%lOa0\$&ujHI(.R

 HC256 code in the March 2025 Comebacker loader

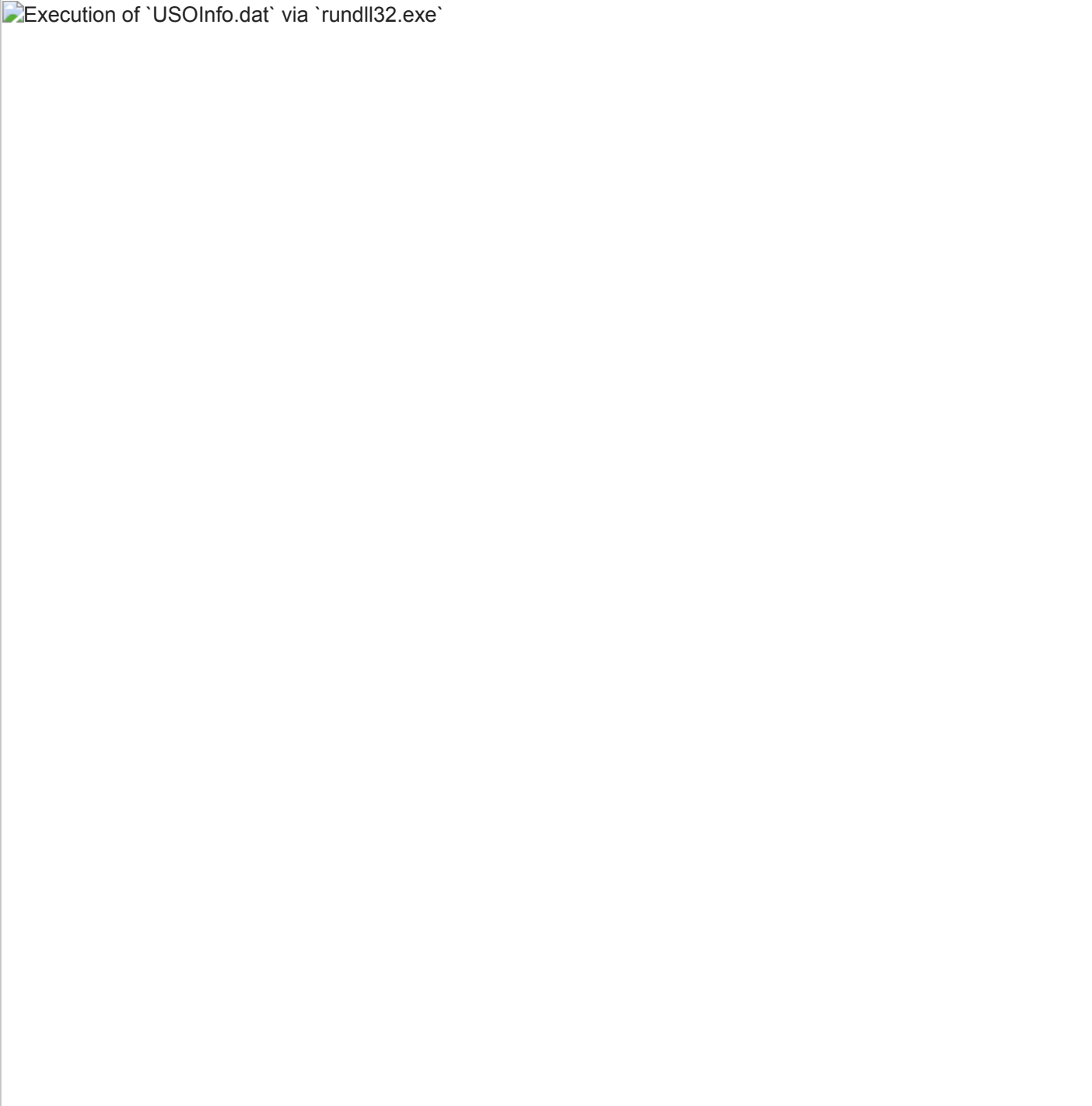
HC256 code in the March 2025 Comebacker loader

 HC256 routine in the Comebacker loader embedded and distributed in PyPI Packages in 2024

HC256 code in a 2024 PyPI-distributed sample

A Python script to decrypt the payload is available in **Appendix C, under “HC256 Decryption Script”**.

After decryption and decompression, the loader writes the next stage to `C:\ProgramData\USOShared\USOInfo.dat` and executes it using `rundll32.exe`. It calls the `GetSysStartTime` export with two arguments: `"dfgdfg"` and `"G3z!X97k7QrwG"`.



Execution of `USOInfo.dat` via `rundll32.exe`

The screenshot shows a Windows command prompt window with the title "Execution of `USOInfo.dat` via `rundll32.exe`". The command prompt is open, and the text "Execution of `USOInfo.dat` via `rundll32.exe`" is visible at the top. The rest of the window is empty, indicating that the execution has completed or is in progress without visible output.

Execution of USOInfo.dat via rundll32.exe

3.2. Comebacker Stage 2 Loader - USOInfo.dat

USOInfo.dat is the in-memory loader for this variant, analogous to **USOPrivate.dll** from the first infection chain. It begins by validating its command-line arguments, checking for **"G3z!X97k7QrwG"**.

If the argument check succeeds, it decrypts its embedded payload. This stage again uses the HC256 stream cipher but with a different, unique key and IV pair.

key, iv: 6w6ZT9|a-0}s\$@;(@&#jPVC4o+V?1IU%

argument value validation and decryption routine

Argument check and HC256 decryption in USOInfo.dat


Following decryption and decompression, the loader loads the final payload into memory. It then calls the payload's **GetWindowSizedW** export with the argument **"3718"**.

The final payload Comebacker, with identical functionality to the Comebacker detailed in Section 2.5.

4. Attack Evolution


4.1. Decryption Process

The Comebacker variant described in the 2021 Google Threat Analysis Group report decrypted its payload using either RC4 or HC256 with the same key and IV.

Decryption code in the Google, Microsoft report's
a75886b016d84c3eaacaf01a3c61e04953a7a3adf38acf77a4a2e3a8f544f855

Decryption code in the Google, Microsoft report's
a75886b016d84c3eaacaf01a3c61e04953a7a3adf38acf77a4a2e3a8f544f855

The variant distributed in 2024 via PyPI packages and the variant discovered in March 2025 consistently used HC256 with identical keys and IVs.

 decryption routine in the jpcert report's comebacker
malware distributed as pycryptoenv

Decryption code in JPCERT's report on Comebacker distributed as pycryptoenv

The newly identified variant deviates from this by introducing a custom XOR/bit-swap algorithm for the initial dropper stage and adopting ChaCha20 for subsequent loader stages.

 ChaCha20 decryption code in the new variant

ChaCha20 decryption code in the new variant

4.2. Communications Encrytion

Past Comebacker variants communicated with their C2 servers in plaintext, including the samples from the 2021 security researcher campaign and the 2024 PyPI campaign.

 jpcert report's comebacker malware distributed as pycryptoenv


Communication code in JPCERT report's report on Comebacker distributed as pycryptoenv

The variants observed since March 2025 introduce encrypted C2 communications, using AES-128-CBC to encrypt C&C traffic.

 newly Comebacker malware AES routine

4.3. Distribution Process


Comebacker, first reported by Google's Threat Analysis Group, was employed in a campaign targeting security researchers with themes of vulnerability research collaboration. The attacker used Visual Studio projects that contained malicious Visual Studio Build Events. In addition, they carried out attacks using an Internet Explorer 0-day. At the time, we published analysis of the exploit on our [blog](#). We have since translated the post to [English](#).



Attacker activity in the 2021 campaign targeting security researchers (Source: Google TAG)

Attacker activity in the 2021 campaign targeting security researchers (Source: Google TAG)

In 2024, the actor published malicious packages to PyPI, using typosquatting tactics to target developers.

 pycryptoconf package used to distribute Comebacker in 2024
(Source: JPCERT/CC)

pycryptoconf package used to distribute Comebacker in 2024 (Source: JPCERT/CC)

While we could not determine the distribution vector for the March 2025 sample, the lure documents from the most recent activity provide strong clues. The documents impersonate specific organizations in the aerospace and defense sector (Edge Group, IIT Kanpur, Airbus) and contain tailored content. This deliberate crafting of decoys for specific targets is a hallmark of spear phishing campaigns aimed at a small set of victims.

5. Conclusion

This report details a recent espionage campaign conducted by the DPRK-nexus threat actor Lazarus Group against the aerospace and defense sectors. The campaign leverages a new variant of the Comebacker backdoor, demonstrating the actor's continued refinement of its malware arsenal.

The actor's use of highly specific lure documents indicates that this is a targeted spear phishing campaign. Although there are no reports of victims so far, the C2 infrastructure remains active at the time of this publication.

Organizations in the aerospace, defense, and research sectors should remain vigilant for phishing attempts and ensure they have robust defenses against macro-based threats.

6. Appendix

Appendix A. MITRE ATT&CK

Tactics	Techniques
Resource Development	T1583.001: Acquire Infrastructure: Domains

Initial Access	T1566.001: Phishing: Spearphishing Attachment					
Execution	T1204.002:User Execution: Malicious File T1204.005: User Execution: Malicious Library T1059.001: Command and Scripting Interpreter: PowerShell T1059.003: Command and Scripting Interpreter: Windows Command Shell T1059.005: Command and Scripting Interpreter: Visual Basic					
Persistence	T1547.001: Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder T1547.000: Boot or Logon Autostart Execution: Shortcut Modification					
Defense Evasion	T1140	Deobfuscate/Decode Files or Information T1027.013	Obfuscated Files or Information: Encrypted/Encoded File T1027.015	Obfuscated Files or Information: Compression T1218.011	System Binary Proxy Execution: Rundll32 T1620	Reflective Code Loading
Command and Control	T1132.001	DataEncoding: Standard Encoding T1573.001	Encrypted Channel: Symmetric Cryptography T1071.001	Application Layer Protocol: Web Protocols T1102	Web Service	

MITRE ATT&CK

Appendix B. IOCs

sha256

- b7d625679fbcc86510119920ffdd6d21005427bf49c015697c69ae1ee27e6bab - docx file
- 046caa2db6cd14509741890e971ddc8c64ef4cc0e369bd5ba039c40c907d1a1f - docx file
- 14213c013d79ea4bc8309f730e26d52ff23c10654197b8d2d10c82bbbcd88382 - docx file
- b357b3882cf8107b1cb59015c4be3e0b8b4de80fd7b80ce3cd05081cd3f6a8ff - docx file
- 7e61c884ce5207839e0df7a22f08f0ab7d483bfa1828090aa260a2f14a0c942c - wpsoffice_aam.cox
- c4a5179a42d9ff2774f7f1f937086c88c4bc7c098963b82cc28a2d41c4449f9e - USOPrivate.dll
- f2b3867aa06fb38d1505b3c2b9e523d83f906995dcdd1bb384a1087b385bfc50 - Comebacker Loader
- 96b973e577458e5b912715171070c0a0171a3e02154eff487a2dcea4da9fb149 - USOInfo.dat

C&C

- hxxps://birancearea[.]com/adminv2
- hxxps://hiremployee[.]com

Open Directory C&C

office-theme[.]com

aes key

x!P<&}mjH2YHRQ',

chacha20 key

ad9c5aca9977d04c73be579199a827049b6dd9840091ffe8e23acc05e1d4a657

HC256 key

- LH*x239udC<*sd_Sej%lOa0\$&ujHl(.R
- 6w6ZT9|a-0)s\$@;(@&#jPVC4o+V?1IU%

Appendix C. Decryption Scripts

Comebacker Dropper Decryption Script

```
def decrypt(enc):
    dec = ''
    for i in enc:
        dec += bin(i ^ 25)[2:].rjust(8, '0')
    enc = dec[:]
    dec = ''
    for i in range(0, len(enc), 4):
        dec += (enc[i + 2] + enc[i + 3] + enc[i] + enc[i + 1])
    enc = dec[:]
    dec = ''
    for i in enc[::-1]:
        dec += '1' if i == '0' else '0'
    final = b''
    for i in range(0, len(dec), 8):
        final += bytes([int(dec[i:i + 8], 2)])
    return final

enc = bytes.fromhex(open('loader_enc', 'r').read())
loader = decrypt(enc)
open('wpoffice_aam.ocx', 'wb').write(loader)

enc = bytes.fromhex(open('lurefile_enc', 'r').read())
lurefile = decrypt(enc)
open('EDGE_Group_Interview_NDA.docx', 'wb').write(lurefile)
```

Comebacker Stage 1 Loader Decryption Script

```
import struct

def rotl32(x, n):
    return ((x << n) & 0xffffffff) | (x >> (32 - n))

def pack4(b):
    return struct.unpack('<I', bytes(b))[0]

def unpack4(x):
    return list(struct.pack('<I', x))

class ChaCha20:
    def __init__(self, key: bytes, nonce: bytes, counter: int = 0):
        assert len(key) == 32
        assert len(nonce) == 12
        self.state = [0] * 16
        self.keystream32 = [0] * 16
        self.counter = counter
        self.position = 64
        self.key = key
        self.nonce = nonce
        self._init_block()
        self._set_counter(counter)

    def _init_block(self):
        constants = b"expand 32-byte k"
        key = self.key
        nonce = self.nonce
        self.state[0] = pack4(constants[0:4])
        self.state[1] = pack4(constants[4:8])
        self.state[2] = pack4(constants[8:12])
        self.state[3] = pack4(constants[12:16])
        for i in range(8):
            self.state[4 + i] = pack4(key[i * 4:(i + 1) * 4])
        self.state[12] = 0
        for i in range(3):
            self.state[13 + i] = pack4(nonce[i * 4:(i + 1) * 4])

    def _set_counter(self, counter):
        self.state[12] = counter & 0xffffffff
        self.state[13] = pack4(self.nonce[:4]) + ((counter >> 32) & 0xffffffff)

    def _quarterround(self, x, a, b, c, d):
        x[a] = (x[a] + x[b]) & 0xffffffff
        x[d] = rotl32(x[d] ^ x[a], 16)
        x[c] = (x[c] + x[d]) & 0xffffffff
        x[b] = rotl32(x[b] ^ x[c], 12)
        x[a] = (x[a] + x[b]) & 0xffffffff
        x[d] = rotl32(x[d] ^ x[a], 8)
        x[c] = (x[c] + x[d]) & 0xffffffff
        x[b] = rotl32(x[b] ^ x[c], 7)

    def _block_next(self):
        x = self.state[:]
        for _ in range(10):
            self._quarterround(x, 0, 4, 8, 12)
            self._quarterround(x, 1, 5, 9, 13)
            self._quarterround(x, 2, 6, 10, 14)
            self._quarterround(x, 3, 7, 11, 15)
```

```

        self._quarterround(x, 0, 5, 10, 15)
        self._quarterround(x, 1, 6, 11, 12)
        self._quarterround(x, 2, 7, 8, 13)
        self._quarterround(x, 3, 4, 9, 14)
    for i in range(16):
        self.keystream32[i] = (x[i] + self.state[i]) & 0xffffffff
    self.state[12] = (self.state[12] + 1) & 0xffffffff
    if self.state[12] == 0:
        self.state[13] = (self.state[13] + 1) & 0xffffffff
        assert self.state[13] != 0

def xor(self, data: bytes) -> bytes:
    output = bytearray()
    keystream = bytearray()
    for word in self.keystream32:
        keystream.extend(struct.pack('<I', word))
    for byte in data:
        if self.position >= 64:
            self._block_next()
            keystream = bytearray()
            for word in self.keystream32:
                keystream.extend(struct.pack('<I', word))
            self.position = 0
        output.append(byte ^ keystream[self.position])
        self.position += 1
    return bytes(output)

key = bytes.fromhex("ad9c5aca9977d04c73be579199a827049b6dd9840091ffe8e23acc05e1d4a657")
iv = bytes.fromhex("edc9ce049daeba35b8687740")

enc = open('USOPrivate_enc', 'rb').read()
dec = b''
cipher = ChaCha20(key, iv, counter=1)
for i in range(0, 0x2D903, 16):
    dec += cipher.xor(enc[i:i+16])
    cipher.state[12] = cipher.state[12] + 1 & 0xffffffff
open('USOPrivate_dec', 'wb').write(dec)

```

Comebacker C&C Data Decryption Script

```

from base64 import b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

enc = b64decode("zTJFPv1/590f+S7AlhXPufXxy5abaigGREwmiXSvF9Q=")
key = b"x!P<&}mjH2YHRQ',"

cipher = AES.new(key, AES.MODE_CBC, key)
dec = unpad(cipher.decrypt(enc), AES.block_size)
print(dec)

```

HC256 Decryption Script

```
from typing import Optional
import logging
import zipfile

def bit_check(n: bytes) -> int:
    return len(n) * 8

def rotr32(x: int, n: int) -> int:
    return (x >> n) + ((x << (32 - n)) & 0xffffffff)

def rotl32(x: int, n: int) -> int:
    return (x << n) + ((x >> (32 - n)) & 0xffffffff)

class HC256():
    def __init__(
        self,
        key: bytes,
        iv: bytes,
        log_level: Optional[int] = logging.WARN
    ):
        logging.basicConfig()
        self.logger = logging.getLogger(__name__)
        self.logger.setLevel(log_level)

        if bit_check(key) != 256:
            self.logger.warning("keylen != 256 bits, null-padding")

        if bit_check(iv) != 256:
            self.logger.warning("ivlen != 256 bits, null-padding")

        key = key + b"\x00" * (32 - len(key))
        iv = iv + b"\x00" * (32 - len(iv))
        # Convert to list of DWORD
        self.key = [
            int.from_bytes(key[i:i+4], byteorder="little")
            for i in range(len(key))[:4]
        ]

        self.iv = [
            int.from_bytes(iv[i:i+4], byteorder="little")
            for i in range(len(iv))[:4]
        ]

        self.logger.debug(f"key: {self.key}")
        self.logger.debug(f"iv: {self.iv}")
        self.__init_cipher()
        self.ctr = 0

    def __init_cipher(self):
        def f1(x):
            return (rotr32(x, 7) ^ rotr32(x, 18) ^ (x >> 3)) & 0xffffffff

        def f2(x):
            return (rotr32(x, 17) ^ rotr32(x, 19) ^ (x >> 10)) & 0xffffffff

        if len(self.key) != 8:
            raise ValueError("Invalid key len!")

        if len(self.iv) != 8:
```

```

        raise ValueError("Invalid IV len!")

self.logger.debug("Setting key/IV")
self.ctr = 0
w = [0] * 2560
self.P = None
self.Q = None

for i in range(len(self.key)):
    w[i] = self.key[i]
    w[i+8] = self.iv[i]

for i in range(16, 2560):
    w[i] = ((
        ((f2(w[i-2]) + w[i-7]) & 0xffffffff) +
        (f1(w[i-15]) + w[i-16]) & 0xffffffff
    ) + i) & 0xffffffff)
self.P = w[512:1536]
self.Q = w[1536:]
if len(self.P) != 1024 or len(self.Q) != 1024:
    raise ValueError(f"P/Q invalid len: P: {len(self.P)}, Q: {len(self.Q)}")
# Run 4096 iters.
self.logger.debug("Running keystream iterations before cipher")
for i in range(4096):
    self.keystream()

def keystream(self) -> int:
    r = None

def g1(x: int, y: int) -> int:
    return ((rotr32(x, 10) ^ rotr32(y, 23)) + self.Q[(x ^ y) % 1024]) & 0xffffffff

def g2(x: int, y: int) -> int:
    return ((rotr32(x, 10) ^ rotr32(y, 23)) + self.P[(x ^ y) % 1024]) & 0xffffffff

def h1(x: int) -> int:
    r = self.Q[x & 0xff]
    r = (r + self.Q[256 + ((x >> 8) & 0xff)]) & 0xffffffff
    r = (r + self.Q[512 + ((x >> 16) & 0xff)]) & 0xffffffff
    r = (r + self.Q[768 + ((x >> 24) & 0xff)]) & 0xffffffff
    return r & 0xffffffff

def h2(x: int) -> int:
    r = self.P[x & 0xff]
    r = (r + self.P[256 + ((x >> 8) & 0xff)]) & 0xffffffff
    r = (r + self.P[512 + ((x >> 16) & 0xff)]) & 0xffffffff
    r = (r + self.P[768 + ((x >> 24) & 0xff)]) & 0xffffffff
    return r & 0xffffffff

j = self.ctr % 1024
j3 = (j - 3) % 1024
j10 = (j - 10) % 1024
j12 = (j - 12) % 1024
j1023 = (j - 1023) % 1024
if self.ctr < 1024:
    self.P[j] = (
        self.P[j] +
        self.P[j10] +
        g1(self.P[j3], self.P[j1023])
    ) & 0xffffffff
    r = h1(self.P[j12])

```

```

        r = (r ^ self.P[j]) & 0xffffffff
    else:
        self.Q[j] = (
            self.Q[j] +
            self.Q[j10] +
            g2(self.Q[j3], self.Q[j1023])
        ) & 0xffffffff
        r = h2(self.Q[j12])
        r = (r ^ self.Q[j]) & 0xffffffff

    self.ctr = (self.ctr + 1) & 0x7ff
    return r

def crypt(self, cipher: bytes) -> bytes:
    uncipher = list(cipher)
    i = 0
    while i < len(cipher):
        k = self.keystream()
        self.logger.debug(f"keystream[{i}]: {k}")
        j = 0
        while (j < 4 and i < len(cipher)):
            uncipher[i] ^= (k & 0xff)
            i += 1
            j += 1
            k = (k >> 8)

    return bytes(uncipher)

key = b'LH*x239udC<*sd_Sej%l0a0$&ujHl(.R'
encdat = open('enc_dat', 'rb').read()
cipher = HC256(key, key)
decdat = cipher.crypt(encdat)
open('PK_dat', 'wb').write(decdat)
zipfile.ZipFile('PK_dat', 'r').extractall('.')

```




엔키화이트햇

ENKI Whitehat

Offensive security experts delivering deeper security through an attacker's perspective.