

Joyce Malave From Boeing Wants a Quote: Inside a 6-Stage Cobalt Strike Campaign With Static Encryption Keys and a Live Filemail Payload

 intel.breakglass.tech/post/boeing-rfq-nkfz5966-cobalt-strike-6-stage-filemail

Breakglass Intelligence

April 1, 2026

BREAKGLASS INTELLIGENCE

Threat Intelligence & Security Research

intel.breakglass.tech

@BreakGlassIntel

A procurement manager at a mid-size industrial supplier opens an email. The sender claims to be Joyce Malave from Boeing, requesting a quotation for a high-quantity order. Attached is a Word document: *Rfq and Payment Schedule.docx*. It looks like money. It's actually a six-layer kill chain that ends with Cobalt Strike running in memory, and the operator's encryption keys hardcoded across every single variant.

This is the story of **NKFZ5966PURCHASE** -- a spear-phishing campaign impersonating Boeing procurement that chains together a Word document, an RTF embedding trick from the early 2010s, JavaScript, PowerShell, a full Python 3.12 runtime, AES-256 encryption, and reflective DLL loading to land a Cobalt Strike beacon without ever touching disk with a recognizable payload. Twenty-two linked samples. Three lure variants. At least one delivery URL still live as of today.

It Started With a Sample

On March 30, security researcher @JAMESWT_WT flagged a suspicious DOCX on X. Within hours, @lowma13 submitted samples to MalwareBazaar under the tag NKFZ5966PURCHASE. By April 1, three distinct lure variants had surfaced -- all sharing identical document metadata,

identical attack chain architecture, and identical encryption keys. The campaign was active and expanding.

GHOST pulled every available sample and began working backward from the payload.

The Lure: Boeing RFQ

The social engineering is straightforward and effective. The emails impersonate "Joyce Malave from BOEING" or "Global Services, LLC," targeting procurement and sales departments with Request for Quotation (RFQ) themed messages. The pitch is simple: *"Give us the best prices as we plan to order for high quantity."*

Three DOCX lure variants were identified:

- **Rfq and Payment Schedule.docx** (first seen March 30)
- **Product_specifications.docx** (first seen April 1)
- **RFQ_PO_ATR29026II.docx** (first seen April 1)

The filenames rotate, but the documents are structurally identical. And buried in the metadata, left unsanitized across every variant:

```
Creator:      Christian Booc
Last Modified By: John
Created:      2021-04-18T19:39:00Z
Modified:     2026-01-10T16:33:00Z
Application:  Microsoft Office Word 16.0
```

The template was created in April 2021. It was weaponized in January 2026 -- nearly five years later. Someone pulled an old document off a shelf, loaded it with a payload, and never bothered to scrub the author fields. "Christian Booc" built the template. "John" armed it. Neither name was removed.

Stage 1: The aFChunk Trick

Open the DOCX and you're looking at a ZIP archive containing OOXML. Standard Office document internals. But buried in the relationships file is an **aFChunk** reference -- an alternative format chunk that tells Word to embed and render an RTF file as if it were part of the document.

The embedded RTF files have randomized names across variants: **rsuas.rtf**, **fetim.rtf**, **reato.rtf**. Each is approximately 4 MB -- enormous for an RTF file. The reason becomes clear when you look inside.

The RTF payload is hex-encoded and hidden inside the **{*\svb}** control word. This is a destination group that Word's RTF parser processes but doesn't render visually. The hex data decodes to an OOXML structure containing an OLE compound document, and inside that OLE

object sits a JavaScript file.

The aFChunk technique isn't new -- it dates back to at least 2017 -- but it remains effective because most email security gateways parse DOCX files at the ZIP level and don't follow aFChunk relationships into embedded RTF content. The hex encoding inside the RTF adds another layer of obfuscation. YARA detections fire on `Embedded_RTF_File` and `FreddyBearDropper`, but these are research rules, not widely deployed production signatures.

Stage 2: The JavaScript Dropper

Three JS variants were extracted from the RTF payloads: `license.js`, `2license.js`, and `3license.js`. Each is approximately 67 KB of obfuscated JavaScript.

The obfuscation scheme is simple but effective. The entire malicious payload is stored as a single concatenated string with the junk separator `tZaVLLetjJ` inserted between meaningful characters:

```
var payload = "WtZaVLLetjJStZaVLLetjJctZaVLLetjJri...";  
var clean = payload.split("tZaVLLetjJ").join("");
```

After deobfuscation, the dropper uses WMI (`winmgmts:root\cimv2`) through `Win32_Process` to spawn the next stage, with `WScript.Shell` handling execution in a hidden window. No user-visible console. No taskbar flash.

Stage 3: PowerShell -- The Download

The JavaScript spawns PowerShell with a Base64-encoded command (UTF-16LE, as is standard for `-EncodedCommand`). Decoded, it reveals a carefully crafted downloader with several notable evasion techniques:

TLS certificate validation is disabled. The script creates a custom `TrustAllCertsPolicy` class that returns `true` for any certificate. This prevents errors when the payload is served from infrastructure with mismatched or self-signed certificates.

AMSI evasion via reflection. Rather than calling `System.Net.WebClient.DownloadFile()` directly -- which AMSI hooks can intercept -- the script uses `Microsoft.VisualBasic.Interaction.CallByName()` to invoke the download method indirectly. This bypasses string-based AMSI detections that look for known WebClient patterns.

Browser impersonation. The download request sets a Chrome 128 User-Agent and a `filemail.com` Referer header, making the traffic appear as a legitimate browser download from the file-sharing service.

The target: a 14.5 MB ZIP file from Filemail.com, downloaded to `%PUBLIC%\Downloads\python312x64.zip`, extracted to `%APPDATA%\Roaming\Templates\python312x64\`, and executed. The ZIP is deleted after

extraction.

The Filemail.com Problem

Filemail.com is a legitimate Norwegian file-sharing service. The attackers use it for payload staging specifically because the domain has clean reputation. URL filtering systems that block unknown or newly-registered domains let `filemail.com` through without a second look.

Three distinct Filemail URLs were identified across campaign variants, each on a different subdomain:

Subdomain	Status
2007.filemail.com	Dead (HTTP 500)
2012.filemail.com	Unknown
2117.filemail.com	LIVE

The live URL on subdomain 2117 serves the payload with a `Content-Disposition` header naming it `545153.mp3` -- an audio file disguise. The actual content is a 14.5 MB ZIP containing a full Python 3.12 distribution. The use of multiple subdomains suggests the operator is either rotating Filemail accounts as they get burned or maintaining parallel delivery infrastructure for redundancy.

As of publication, the 2117 URL remains active.

Stage 4: A Whole Python Runtime

The extracted ZIP contains 975 files. The centerpiece is a legitimate, signed `pythonw.exe` -- Python 3.12, windowless variant. Alongside it: the PyCryptodome library (for AES operations), a loader script called `Protected.py`, a helper module `_builtin_.py`, and a file named `license.pdf`.

That PDF isn't a PDF.

`license.pdf` is a 743 KB encrypted DLL. No PDF header, no document structure -- just raw encrypted bytes with the `.pdf` extension. The naming is deliberate: endpoint detection systems that inspect file extensions will see a benign document type. Systems that check magic bytes would flag it, but the file is never opened by a PDF reader. It's only read by `Protected.py`.

The Python runtime itself is completely legitimate. It would pass any application whitelisting that trusts signed Python binaries. The malicious logic lives entirely in the Python scripts, which don't contain any obviously malicious imports -- just `Crypto.Cipher.AES`, `base64`, `zlib`, and standard library modules. The real tradecraft is in how the scripts use these tools.

Stage 5: Four Layers of Obfuscation, Then AES

`Protected.py` is 392 lines and 62 KB. Its string obfuscation follows a fixed pipeline:

1. **Base64 decode** the embedded blob
2. **zlib decompress** the result
3. **Reverse** the byte order
4. **ROT13** the string
5. **XOR** with key value 16

After deobfuscation, the script reads `license.pdf`, decrypts it with AES-256-CBC, applies a multi-key XOR operation, and reflectively loads the resulting DLL into memory.

The encryption keys, extracted from the deobfuscated script:

```
AES-256 Key: a34c243e8fae4a20ad13a0e6be19749c9b0ba47ac8e79af0e4da57f59373c003
AES IV:      ad4b127c683d974dbeab87331b864884
XOR Key 1:   aab57c82934f7296a1cf7e8095eca567
XOR Key 2:   811ea130857d3fd0c8d32f8b401518e9
```

These keys are **identical across all analyzed variants**. The operator is not generating per-target or per-sample keys. Every deployment of this campaign uses the same AES key, the same IV, the same XOR keys. This is a significant OPSEC failure -- it means a single key extraction breaks every past and future sample that uses the same `Protected.py` loader.

Stage 6: Cobalt Strike in Memory

The decrypted DLL never touches disk. It's loaded reflectively -- mapped into the Python process's memory space and executed without being written to the filesystem. This bypasses file-based antivirus scanning entirely.

YARA analysis of memory dumps extracted from sandbox execution confirmed the final payload. Five memory regions were dumped, and two of them matched the `cobalt_strike_tmp01925d3f` signature. The other dumps showed anti-debugging code, .NET components, TLS callback structures, and a PE without an import table (a hallmark of reflectively loaded executables).

The complete in-memory artifact set:

Dump	Size	Key Finding
memorydump 1	811 KB	PE with no import table (reflective load)
memorydump 2	73 KB	.NET components
memorydump 3	716 KB	Anti-debugging, digital certificate structures

Dump	Size	Key Finding
memorydump 4	733 KB	Cobalt Strike
memorydump 5	876 KB	Cobalt Strike

From email attachment to Cobalt Strike beacon, the entire chain executes without dropping a single flagged binary. Every stage uses either a legitimate application (Word, wscript, PowerShell, pythonw.exe) or operates entirely in memory.

Persistence: Hiding Behind Realtek Audio

The campaign establishes persistence through a registry Run key designed to look like a Realtek audio service:

```
Key: HKCU\Software\Microsoft\Windows\CurrentVersion\Run
Value: RtkAudUService
```

The value name mimics Realtek's legitimate **RtkAudUService64.exe** audio manager, which appears on millions of Windows machines. A defender scrolling through Run keys would likely pass over it.

But the execution method is what makes this interesting. Rather than pointing the Run key directly at **pythonw.exe**, the operator uses **SyncAppvPublishingServer.vbs** -- a legitimate Windows LOLBin (Living Off the Land Binary) that ships with App-V virtualization. This is LOLBAS technique T1218: the VBS script is signed by Microsoft and trusted by application whitelisting policies. It acts as a proxy to launch the actual malicious command:

```
SyncAppvPublishingServer.vbs n;
$e=Join-Path $env:USERPROFILE 'AppData\Roaming\Templates\python312x64';
$p=Join-Path $e 'pythonw.exe';
$s=Join-Path $e 'Protected.py';
if((Test-Path $p)-and(Test-Path $s)){Start-Process $p $s -WorkingDirectory $e}
```

A Microsoft-signed VBS script launches PowerShell, which launches a legitimate signed Python binary, which loads an encrypted DLL into memory. Three levels of signed-binary indirection between the registry key and the actual malware.

The OPSEC Failures

For all its technical sophistication, this campaign leaks intelligence at every turn:

"Christian Booc" and "John." The document metadata was never sanitized. These names -- whether real or aliases -- are consistent across all variants and provide a starting point for attribution.

A five-year-old template. Creating the document in 2021 and weaponizing it in 2026 suggests template reuse from a builder toolkit. The operator likely has a library of pre-made lure documents.

NKFZ5966PURCHASE. The campaign identifier appears in the original source DOCX filename and was used as the MalwareBazaar tag. Whether this is an internal tracking code or an artifact of the builder, it links every sample in the campaign together.

Hardcoded, static encryption keys. The same AES key, IV, and XOR keys across all variants means that once one sample is cracked, every sample in the campaign is cracked. Key rotation costs nothing. Not rotating is pure laziness.

Filemail.com account reuse. Multiple URLs from the same service -- and one still live -- provide network-level detection opportunities and give abuse teams a clear target for takedown.

The Full Campaign: 22 Samples

MalwareBazaar's NKFZ5966PURCHASE tag links 22 samples spanning every stage of the kill chain. Three DOCX lures, three JavaScript droppers, one PowerShell downloader, the Python loader, the encrypted DLL, and five memory dumps from the final Cobalt Strike payload. The campaign is actively expanding -- new variants appeared as recently as April 1.

Indicators of Compromise

File Hashes (SHA256)

DOCX Lures:

6ad6c38552304b963d6a53e77078c6741cbebf52e758716c470be92c79805cb4	Rfq and Payment
Schedule.docx	
20cff974367eed6e5b208d69ed49e7a9f50afbeeb60cf2f23a3a2e4ca3f1e08c	
Product_specifications.docx	
b7077463eec3d4107f1fcaa7a00847f0921f38ce018221b553e06c1861458ee2	RFQ_PO_ATR29026II.docx
5dfca625893c0215fd5aabc7be6dfdc7534fd579e2e72a7e58e14dbb3750097c	
NKFZ5966PURCHASE_ORDER_SOURCE.docx	

JavaScript Droppers:

2927bd11ed8d3fbadf7cb3960edf1cd30d1cf515853cb9c0fcad42fabce745d8	license.js
0794add65a271388acc6ab87a0dc2fe47373b40921f22dec12c02f74fbe6b154	2license.js
b0e20b5136c9d7ee37bb7c9e044e46f4a29049038ec3543156c1e84c7bd6f062	3license.js

PowerShell Downloader:

bba584c9c26bfe14083256f4f2ec9ea6bcf12db3cf7e1b7424f90fcccced508be

Python Loader:

2f515997ab1c7f5ab94a46041ad2af06031a842469b65bcbdd2c64bd47f12a896 Protected.py

Encrypted DLL:

d3e13175378035d36ff5e568748e1b063f4216e077516ffa79683ddb43ed7524 license.pdf

In-Memory Cobalt Strike:

6601cc35931924371842d26b6ac6abd770b1924916e0fd10756a61dcf8afd882 memorydmp
fceab88e7ebbf3e22350818c11ec7c26afaa97eec27418cdaa193c5551ccebf5 memorydmp
6242051c4bccec96cd7703c7387bbf31deb6fac3d8e6e0f88de287edea153653 memorydmp
d41757c87c22597f4d14406a356b50022cb9a6dcdd9baf0b7075d4fcff3bf774 memorydmp (Cobalt Strike)
3aa4815ddc6260b5523a85c2a1746bdb919959a3f438a1487e2cb1a668302bcc memorydmp (Cobalt Strike)

Network Indicators

hxxps://2007[.]filemail[.]com/api/file/get?
filekey=53VkmZ8l67SvighkrcVFIawFCS9tbjRQNSlziw1sS8FshApkve0_aRg5y3k
hxxps://2012[.]filemail[.]com/api/file/get?
filekey=3Bc1FrI2tmhg1RY2vtfsjD5Qrh25soto4juqCH1r90a_ULucj50qm0eQGc
hxxps://2117[.]filemail[.]com/api/file/get?
filekey=1WnIxXZfRqQG9ynciDtjti0l_Ezx18Dk52UsvBwx-Xttb_ZUYBA0gPLu7_g

Behavioral Indicators

File Paths:

%PUBLIC%\Downloads\python312x64.zip
%APPDATA%\Roaming\Templates\python312x64\
%APPDATA%\Roaming\Templates\python312x64\pythonw.exe
%APPDATA%\Roaming\Templates\python312x64\Protected.py
%APPDATA%\Roaming\Templates\python312x64_builtin_.py
%APPDATA%\Roaming\Templates\python312x64\license.pdf

Registry:

HKCU\Software\Microsoft\Windows\CurrentVersion\Run\RtkAudUService

User-Agent:

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.0.0 Safari/537.36

Extracted Encryption Keys

AES-256 Key: a34c243e8fae4a20ad13a0e6be19749c9b0ba47ac8e79af0e4da57f59373c003
AES IV: ad4b127c683d974dbeab87331b864884
XOR Key 1: aab57c82934f7296a1cf7e8095eca567
XOR Key 2: 811ea130857d3fd0c8d32f8b401518e9

MITRE ATT&CK

Tactic	Technique	ID
Initial Access	Spear-Phishing Attachment	T1566.001
Execution	Malicious File	T1204.002
Execution	JavaScript	T1059.007
Execution	PowerShell	T1059.001
Execution	Python	T1059.006
Execution	System Binary Proxy Execution	T1218
Defense Evasion	Obfuscated Files	T1027
Defense Evasion	Masquerading	T1036
Defense Evasion	Reflective Code Loading	T1620
Defense Evasion	Deobfuscate/Decode	T1140
Persistence	Registry Run Keys	T1547.001
Command and Control	Web Protocols	T1071.001
Command and Control	Ingress Tool Transfer	T1105

Detection Guidance

For email security teams: Hunt for DOCX attachments with Boeing/RFQ themes. Check for aFChunk relationships in OOXML structure -- any DOCX containing an embedded RTF via aFChunk deserves inspection. The document metadata (Creator: "Christian Booc") is a direct signature for this campaign.

For endpoint teams: Search for RtkAudUserService in HKCU\...\Run keys. Hunt for pythonw.exe executing from %APPDATA%\Roaming\Templates\. Monitor for SyncAppvPublishingServer.vbs spawning PowerShell -- this LOLBin has very few legitimate use cases on standard workstations.

For network teams: Block the three Filemail.com payload URLs. Consider blocking *.filemail.com/api/file/get paths entirely if the service isn't business-critical. The Chrome 128 User-Agent combined with a filemail.com Referer from a non-browser process is a high-fidelity detection.

For threat hunters: The encryption keys are static. Any sample using the same AES key (`a34c243e...c003`) belongs to this campaign, even if the DOCX lures change. YARA rules targeting the `tZaVLLetjJ` JavaScript separator string or the specific Base64-zlib-reverse-ROT13-XOR deobfuscation chain in Python will catch variants.

Takeaways

This campaign is technically competent but operationally sloppy. The attack chain is well-designed: six stages, each using legitimate tools or in-memory execution, with the final payload never touching disk. The aFChunk RTF embedding technique evades most email scanners. The Python 3.12 runtime provides a trusted execution environment. The AES+XOR encryption protects the payload at rest. The LOLBin persistence avoids application whitelisting.

But the operator reuses everything. Same document metadata. Same encryption keys. Same file-sharing service. Same campaign tag. Same builder. A defender who cracks one sample has cracked every sample in the campaign. The sophistication is in the tooling, not the tradecraft -- which suggests a purchased or shared attack framework being operated by someone who doesn't fully understand (or doesn't care about) operational security.

The live Filemail URL is the immediate action item. Every hour it stays up is another potential victim.

Initial samples identified by [@JAMESWT_WT](#) and TomU. Campaign analysis and full attack chain reconstruction by GHOST. All samples available on [MalwareBazaar](#) under tag `NKFZ5966PURCHASE`.

Update: Deep Attribution Dive (April 1, 2026)

After initial publication, our GHOST investigation system completed a deep attribution analysis. Key additions:

Cobalt Strike Confirmed

Memory dump analysis confirmed Cobalt Strike via source file references (`httpdnld.cpp`, `persistappdata.cpp`, `httpresource.cpp`, `create_process_server.cpp`) and YARA rule match `cobalt_strike_tmp01925d3f`. The DLL contains a custom `.nep` (Neptune) PE section with JMP thunks for a specialized loader.

Anti-Forensics

- **Timestomped to 2045-02-16** -- deliberate future date to confuse timeline analysis
- **Swapped key labels** -- the variable named `xor_key1` is actually the AES key, and `aes_key` is actually the XOR key. Designed to mislead analysts.
- **DLL export name IEX** -- mimics PowerShell Invoke-Expression

Builder Tool Confirmed

All 3 RTF payloads share an identical **210,673-byte prefix** before diverging at the JS payload injection point. Three different JS separators, three Filemail accounts, three RTF filenames -- all from the same automated builder.

Expanded Targeting

- **Italian organizations** confirmed as secondary targets (Spam-ITA tags on MalwareBazaar)
- **Two payload URLs still LIVE** as of April 1: 2117.filemail.com and 2012.filemail.com

Remaining Gaps

- CS watermark: beacon config is runtime-injected, needs dynamic sandbox
- C2 server: encrypted in beacon, needs ANY.RUN submission
- "Christian Booc": no OSINT hits, likely fabricated

Additional IOCs

- Memory dump (CS DLL):
d41757c87c22597f4d14406a356b50022cb9a6dcdd9baf0b7075d4fcff3bf774
- Filemail URL: hxxps://2012[.]filemail[.]com/api/file/get?
filekey=3Bc1FrI2tmhg1RY2vtfsjD5Qrhm25soto4juqCH1r9Oa_ULucj5OqmOeQGc
- PE section: .nep (Neptune), Export: IEX, Timestamp: 2045-02-16 (timestomped)

Updated post-publication with GHOST attribution findings.

Share 

Related Reports

[high](#)

[Multi-RAT Operation Dismantled: WaterHydra APT Nexus, Five AES Keys Recovered, and Live C2 Infrastructure Mapped Across Three Continents](#)

[3/7/2026](#)