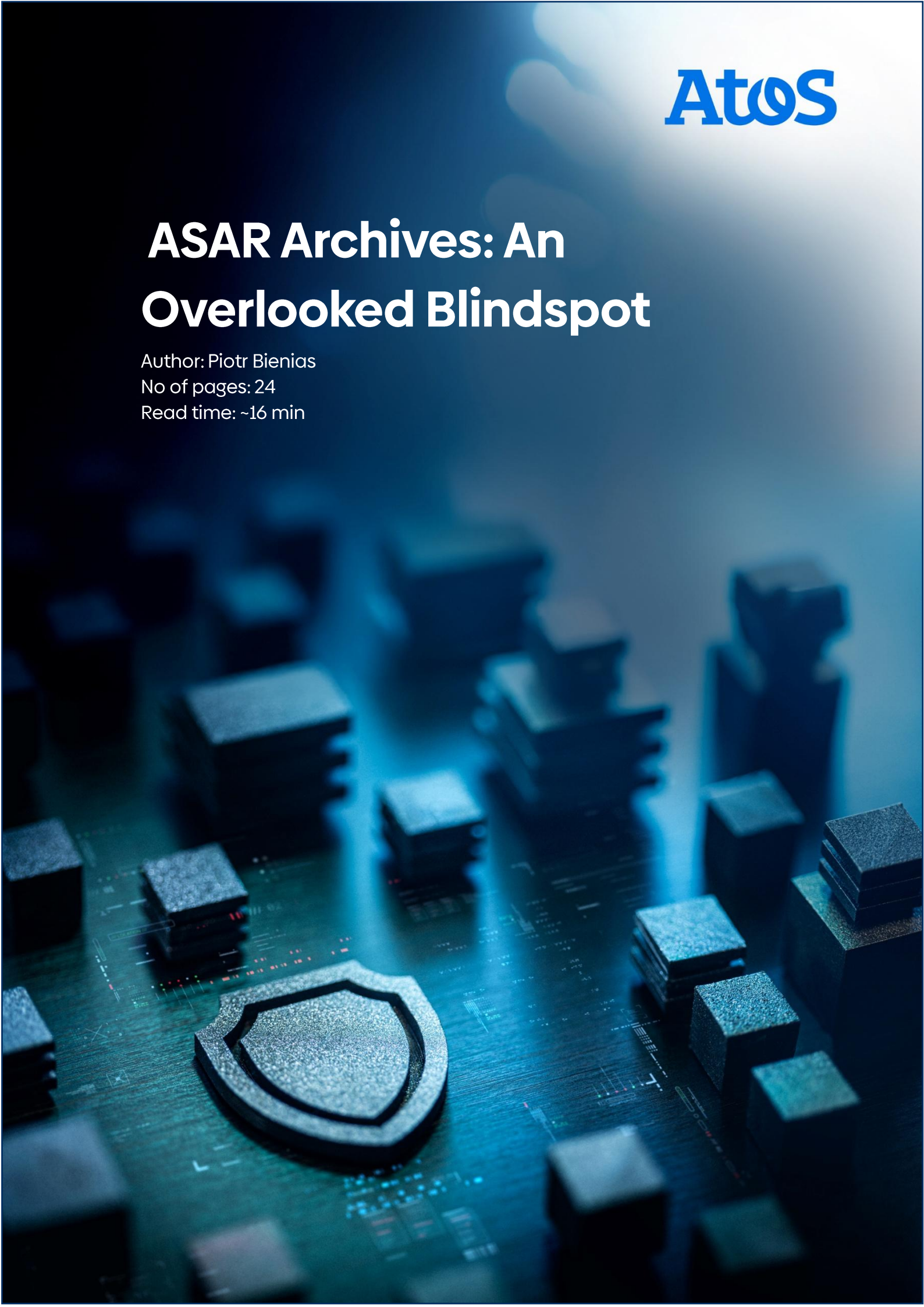


ASAR Archives: An Overlooked Blindspot

Author: Piotr Bienias
No of pages: 24
Read time: ~16 min

Contents

1 Prelude

This research was directly inspired by Atos's recent ClickFix variation analysis. During the investigation it was noticed that first payload dropped by the malicious script is not any crafty malware, but a legitimate application that no security vendor marked as a threat according to VT scanning. Despite this, after execution said application acted as a C2 beacon and deeper analysis revealed that attackers exploited the way that Electron applications are storing their logic, hidden inside ASAR archive. Compiled applications remain intact regardless of its archived content being modified, allowing attackers to bypass signature-based detection.

2 Summary

ASAR archives are a fundamental part of Electron applications and store core application logic in plaintext within a single file that is executed transparently by the runtime. Because Electron does not enable antitampering protection by default, the contents of an ASAR archive can be modified without altering the signed executable, enabling attackers to run malicious code under the context of legitimate applications. Many antivirus engines fail to consistently inspect ASAR contents due to the format's lack of a recognizable magic signature, often treating it as opaque data rather than a structured archive. This results in reduced static detection rates for malicious scripts embedded inside ASAR files, as demonstrated by both synthetic and real-world examples. While well-known malware may still be blocked at execution time, less distinctive payloads can evade detection, highlighting ASAR archives as a visibility gap that favors attackers unless integrity protections and behavior-based monitoring are applied.

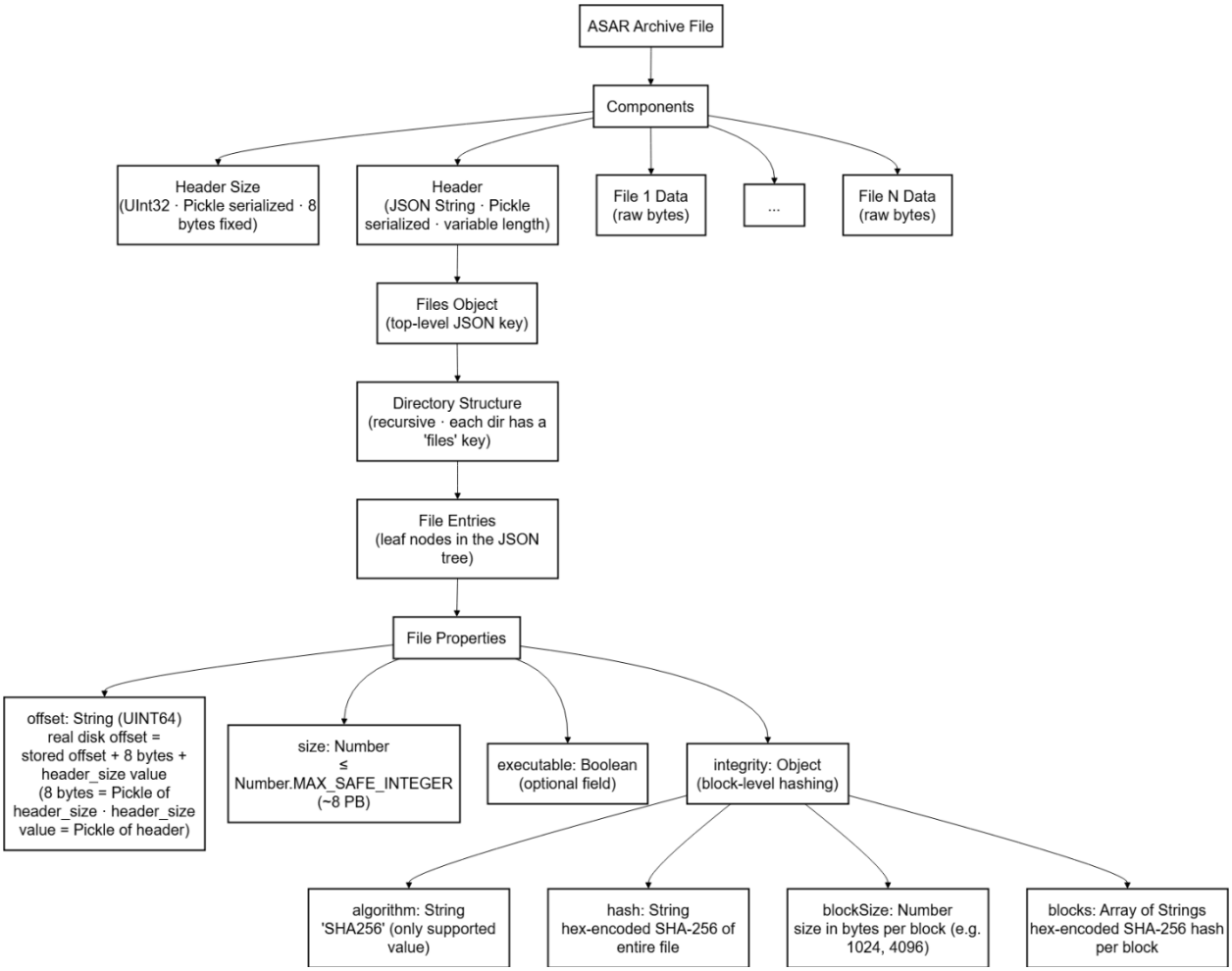
3 Motivation

Electron (formerly known as Atom Shell) is an open-source framework used for developing cross-platform applications with web technologies like HTML, CSS and JS. It has been around since 15 July 2013 and by now is backbone of many widely used applications like Visual Studio Code, GitHub Desktop, Microsoft Teams, Discord, TIDAL, 1Password, OpenVPN Connect and many, many more. Based on Discord and Teams alone we are talking about close to 1B users. Main

goal of this research is to spread awareness about potential risks that may come with Electron-based applications, highlighting the current detection gap existing in modern AV solutions and providing a set of suggestions on how to protect help to monitor end protect your environments against those threats.

4 ASAR archives

ASAR (Atom Shell Archive) is a custom archive format designed specifically for Electron applications to bundle application resources (JavaScript, HTML, CSS, assets) into a single file, named "*app.asar*". It was introduced to simplify application distribution and reduce filesystem overhead.



ASAR archive structure based on "https://deepwiki.com/electron/asar"

ASAR is a tar-like container that concatenates files without compression, meaning all file contents are stored in their original form inside the archive. The archive starts with a binary header, followed by raw file data appended sequentially.
The header contains a JSON metadata structure describing:

- Directory hierarchy
- File offsets
- File sizes
- Executable flags
- Optional integrity metadata (hashes)

ASAR supports random access to individual files without extracting the entire archive, thanks to file data offsets stored as 64-bit values represented as strings in JSON.
Electron patches Node.js filesystem APIs so that ASAR archives are treated as virtual directories, allowing calls such as fs.readFile, fs.readdir, and require() to work transparently on ASAR paths.
ASAR archives are read-only at runtime; filesystem APIs that attempt to modify files (write, delete, chmod) do not work on ASAR contents.

5 Electron applications logic

When an Electron app is launched, the binary looks inside "*resources/*" subdirectory for app code in priority order: "*app.asar*" first, then "*app/*", then "*default_app.asar*" as a fallback.

Name	Date modified	Type	Size
locales	4/10/2026 11:32 AM	File folder	
resources	4/10/2026 11:32 AM	File folder	
chrome_100_percent.pak	4/10/2026 11:32 AM	PAK File	151 KB
chrome_200_percent.pak	4/10/2026 11:32 AM	PAK File	230 KB
d3dcompiler_47.dll	4/10/2026 11:32 AM	Application extens...	4,802 KB
electron-date-app.exe	4/10/2026 11:32 AM	Application	172,134 KB
ffmpeg.dll	4/10/2026 11:32 AM	Application extens...	2,799 KB
icudtl.dat	4/10/2026 11:32 AM	DAT File	10,467 KB
libEGL.dll	4/10/2026 11:32 AM	Application extens...	468 KB
libGLESv2.dll	4/10/2026 11:32 AM	Application extens...	7,513 KB
LICENSE.electron.txt	4/10/2026 11:32 AM	Text Document	2 KB
LICENSES.chromium.html	4/10/2026 11:32 AM	Chrome HTML Do...	8,960 KB
resources.pak	4/10/2026 11:32 AM	PAK File	5,158 KB
snapshot_blob.bin	4/10/2026 11:32 AM	BIN File	300 KB
v8_context_snapshot.bin	4/10/2026 11:32 AM	BIN File	664 KB
vk_swiftshader.dll	4/10/2026 11:32 AM	Application extens...	5,188 KB
vk_swiftshader_icd.json	4/10/2026 11:32 AM	JSON Source File	1 KB
vulkan-1.dll	4/10/2026 11:32 AM	Application extens...	932 KB

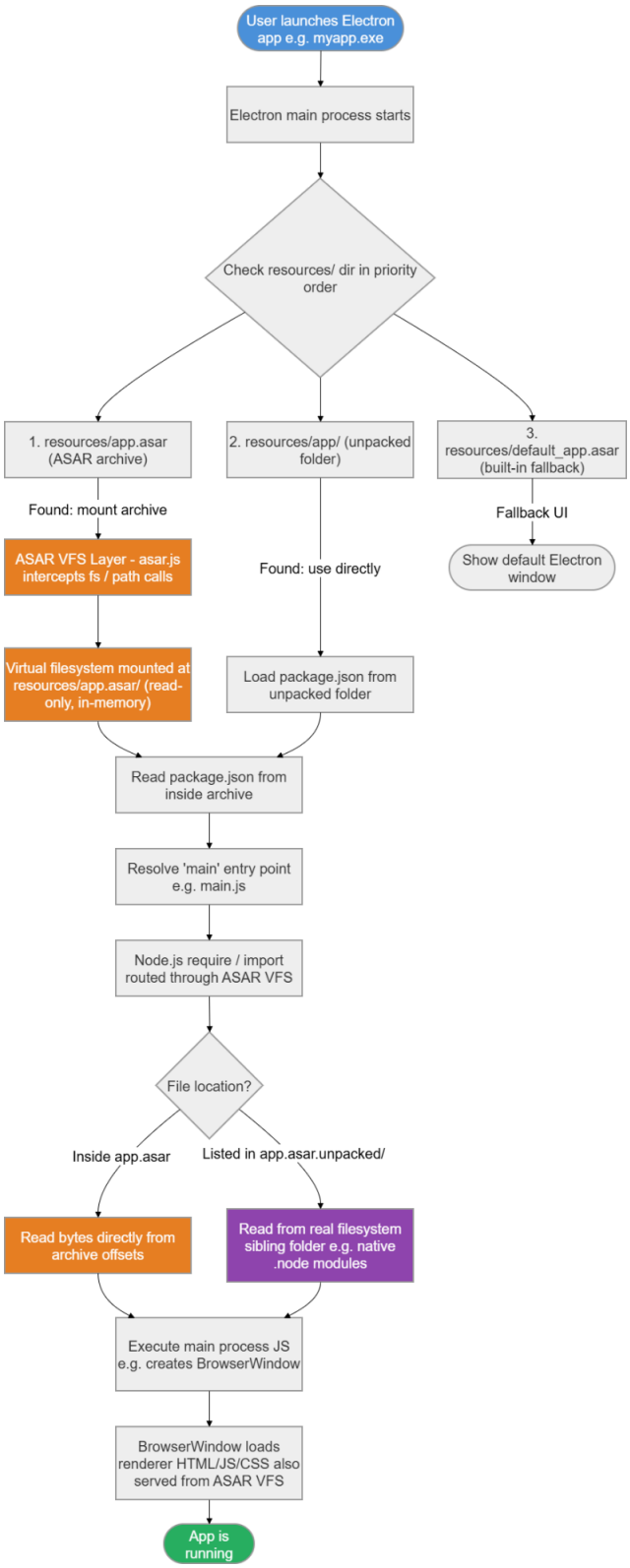
Screenshot of a sample Electron application file structure

Name	Date modified	Type	Size
app.asar	4/10/2026 11:32 AM	ASAR File	2,161 KB

Content of Electron application "resources" subdirectory

If "*app.asar*" is found, Electron mounts it as a virtual filesystem by patching Node's fs APIs - all file reads are transparently redirected into the archive by byte offset, so the app code never knows it's running from inside a single file. It then reads "*package.json*" from the (real or virtual) filesystem to find the main entry point, loads it via require, and the main process starts up - creating

BrowserWindow instances and loading renderer HTML/JS/CSS, which are also served through the same ASAR virtual filesystem.
Files that cannot run from inside the archive (e.g. native .node modules) are placed in the "*app.asar.unpacked/*" sibling directory and read from the real filesystem instead.



Graph showing Electron application logic

Be default there are no antitampering mechanisms implemented in electron app distributions. This means that anyone can modify application logic hidden inside the "*app.asar*" archive and execute it with said application while keeping its certificate intact. In its core it is a bypass mechanism very similar to DLL sideloading, where legitimate applications are running malicious code hidden inside additional payload.

6 Historical exploitation of ASAR archive

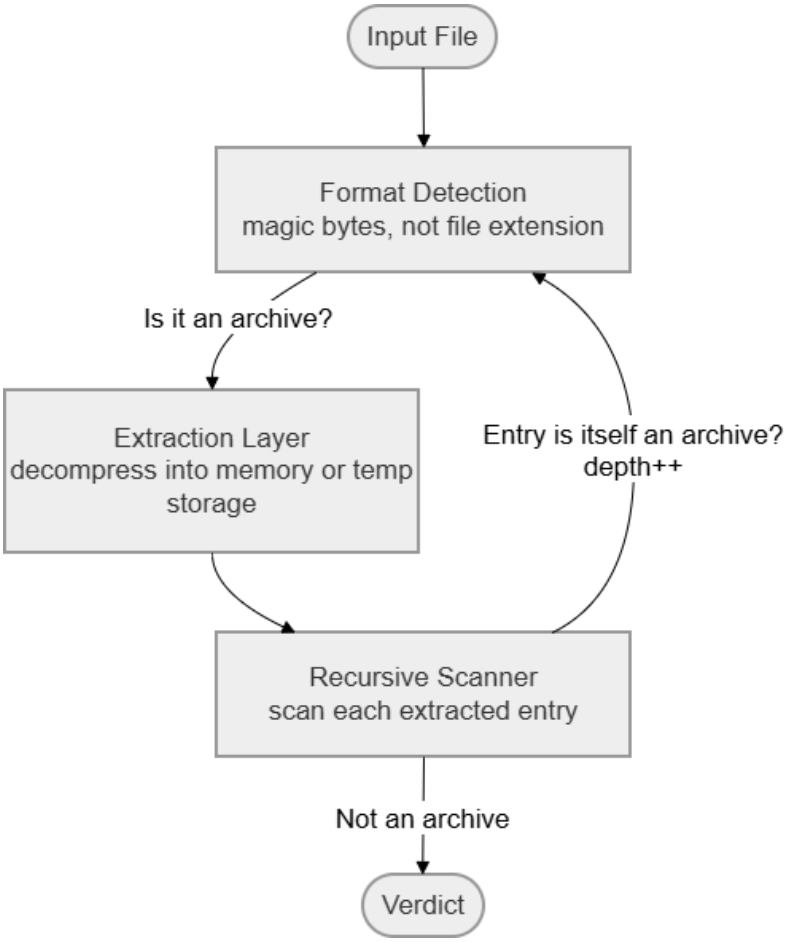
Beyond malicious WorkFlowy instance observed by us in one of unique ClickFix campaign in February 2026, the exploitation of the ASAR archive has grown in popularity as a stealthy persistence and credential theft vector across diverse malware campaigns between 2024 and early 2026. A significant trend involves the targeted manipulation of cryptocurrency wallets, such as Exodus and Atomic, where the Pentagon Stealer and Hexon Stealer campaigns utilized command servers to overwrite local app.asar files with patched versions designed to exfiltrate mnemonic phrases and passwords. Similar persistence mechanism was described in Trend Micro's analysis of stealers abusing GitHub Codespaces to target Discord users, where malware patched the index.js file within the discord_desktop_core package found inside the app.asar archive. This technique was mirrored in the gaming sector throughout 2025, where researchers identified stealer families like Leet and Sniffer Stealer masquerading as gaming utilities by modifying app.asar to harvest Discord tokens and browser data. Further sophistication was observed in the Water Curse campaign, which utilized a "Clone, Compile, Compromise" strategy on GitHub to deliver the DULLRAT backdoor - a malicious script hidden within the archive's "main.js" to perform system discovery and anti-debugging.

7 AV Engines and archive scanning

Modern AV engines have their own proprietary detection mechanism, but when it comes to archives, they all generally have the same procedure used while scanning:

1. Engines identify the archive type by magic bytes (file signatures), not the file extension, because extensions can be spoofed.
2. The engine decompresses archive content, typically:
   - In-memory for small files (fast, no disk I/O)
   - To a temp sandbox directory for large or nested archives
   - Streaming for formats that support it (GZip, BZip2)
3. Each extracted entry is scanned independently and gets the same treatment as a top-level file:
   - Hash matching — MD5/SHA256 of the file compared against known-malware databases
   - Byte signature matching — pattern/string matching against the file's raw bytes
   - YARA rules — more expressive pattern rules (byte sequences, conditions, offsets)
   - Heuristics — structural anomalies, suspicious API imports (PE files), obfuscation indicators
   - ML model scoring — many modern engines pass features to a classifier
   - Emulation / sandboxing — execute code in a micro-VM to observe behavior (e.g., macros, scripts)

   If an entry inside the archive is itself a ZIP/RAR/etc., the engine pushes it back through Step 1 (depth counter incremented).
4. If any file inside the archive is flagged
   - The archive itself is reported as infected
   - Specific path is included: e.g., archive.zip » inner.zip » payload.exe
   - Engine may quarantine the outer archive or just block delivery



AV scanner archive scanning flow

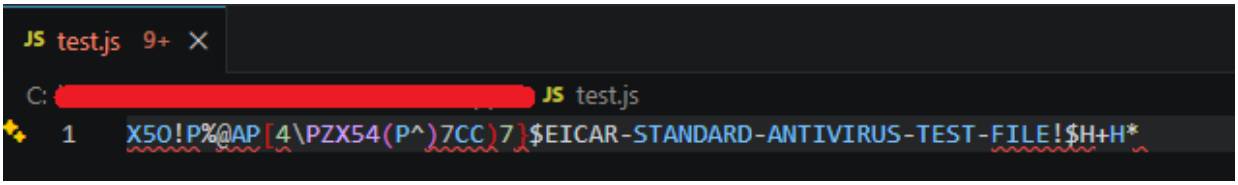
But the engines don't just blindly go down the rabbit hole regardless of the archive size, amount of the files or nested layers. Limits are enforced to prevent various threats this may pose. For instance, ClamAV, an open-source AV engines has following hard limits:

Limit	ClamAV Default	Purpose
Max recursion depth	15	Preventing deeply nested bombs
Max decompressed file size	100 MB	Per-file cap
Max total scan size	400 MB	Entire archive cap
Max number of files	10,000	Preventing file-count bombs

As format identification is driven primarily by magic bytes, archive types that lack a well-defined or widely recognized signature are handled inconsistently by AV engines. This directly affects ASAR archives, which do not begin with a fixed signature that uniquely identifies them as a container format. During scan, such files are commonly treated as opaque raw data strings rather than logical archives, preventing recursive enumeration of embedded files. In such cases, engines rely on broad content inspection techniques such as scanning for byte patterns, embedded strings, and entropy anomalies rather than per-file analysis of the archive's contents. As a result, plaintext application logic stored inside ASAR archives may bypass conventional archive scanning workflows entirely, creating a detection gap that exists independently of compression, encryption, or obfuscation and is inherent to how ASAR archives are identified and processed by AV engines.

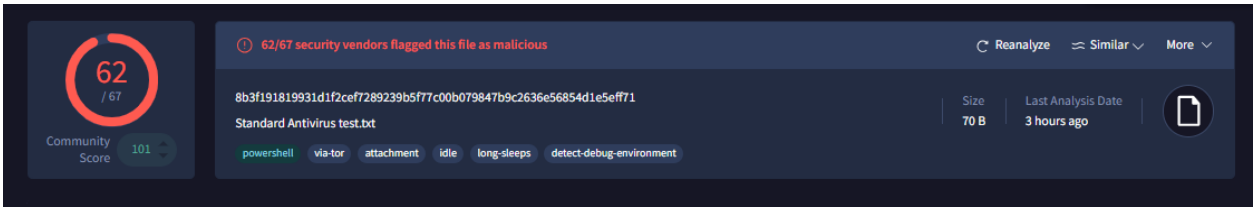
8 ASAR vs. AV engines

To test how well various AV engines handle ASAR format a simple test was prepared. JavaScript file was created, containing only EICAR string, and scanned with Virus Total.



Test.js containing EICAR string

Not surprising, 62 out of 67 available security vendors flagged the file as malicious

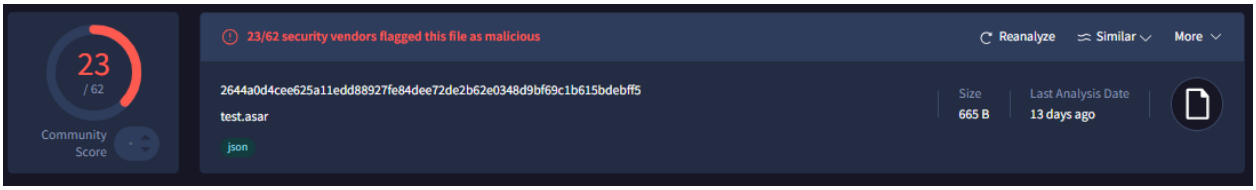


Scanning results of unpacked JS from VirusTotal

Then the same JS was packed into "*test.asar*" archive and scanned again.

Screenshot of “test.asar” content

As visible on above screenshot, any simple text editor can open prepared archive and see the EICAR string in plaintext. However, this time only 23 out of 62 vendors were able to detect the threat.

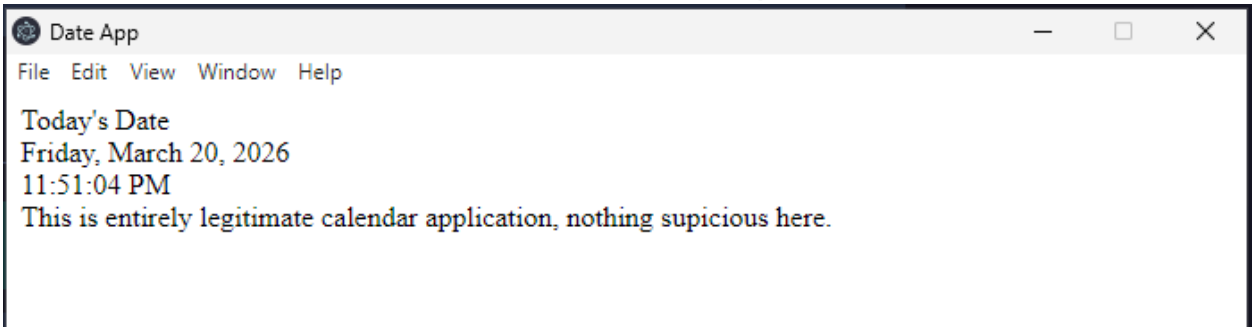


Scanning results of "test.asar" archive from VirusTotal

Links to both samples are available here: “test.js”, “test.asar”.
This little experiment shows huge gap in ASAR handling capabilities of AV engines. It is important to keep in mind that results available via VirusTotal refer to static analysis and it is entirely possible, that engines that missed the threat during static analysis will be capable of preventing execution of malicious code from ASAR archive, but it delays potential detection giving attackers more time to act on their objective.

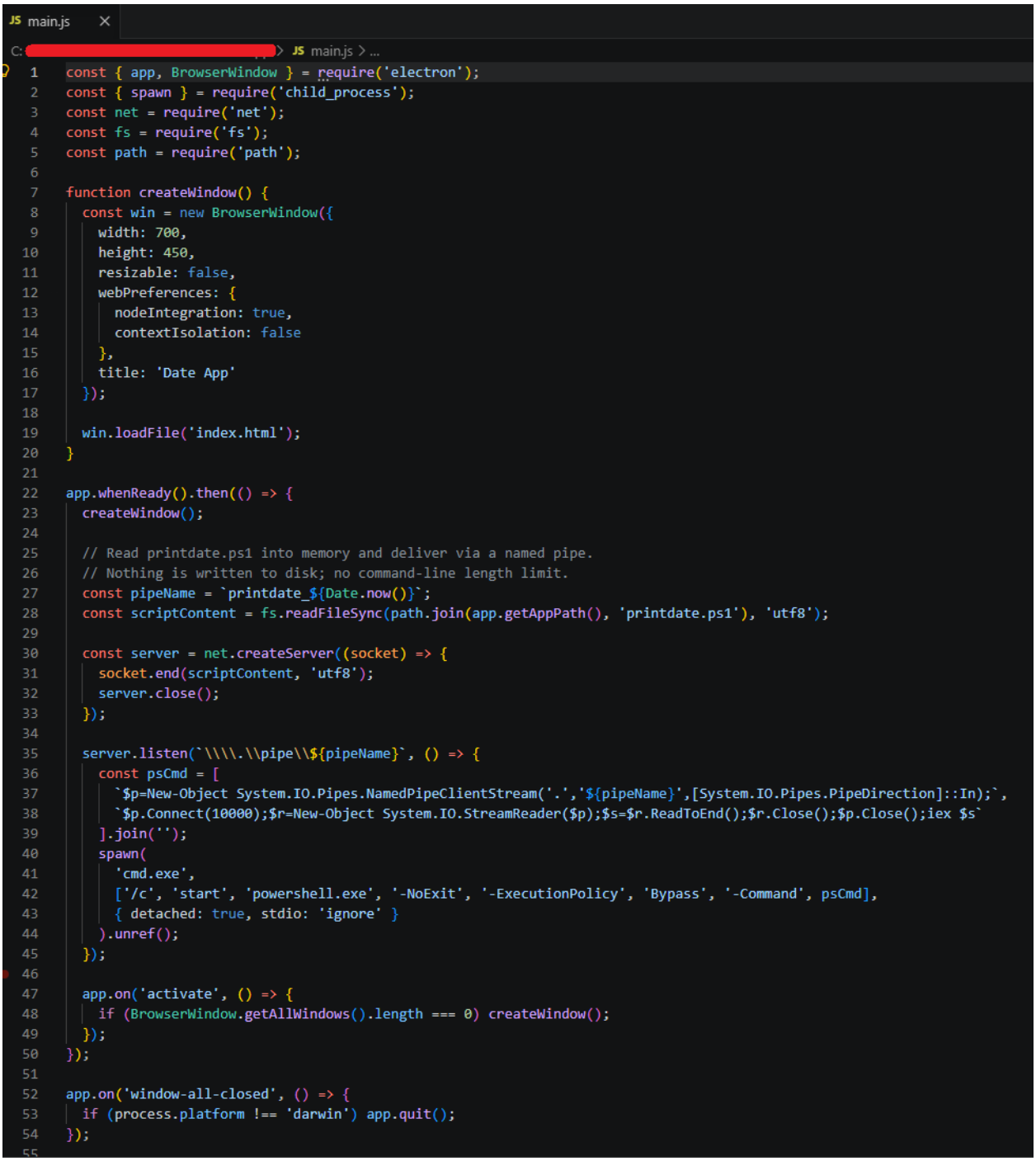
9 Behavioral tests

It was our goal to check if we are able to detect the execution of potentially malicious content from Electron based applications, regardless of whether they are signed or not. For this purpose, a simple Electron application was created, showing the current date and time upon execution. When the app starts, it opens a 700×450 Electron window loading index.html.



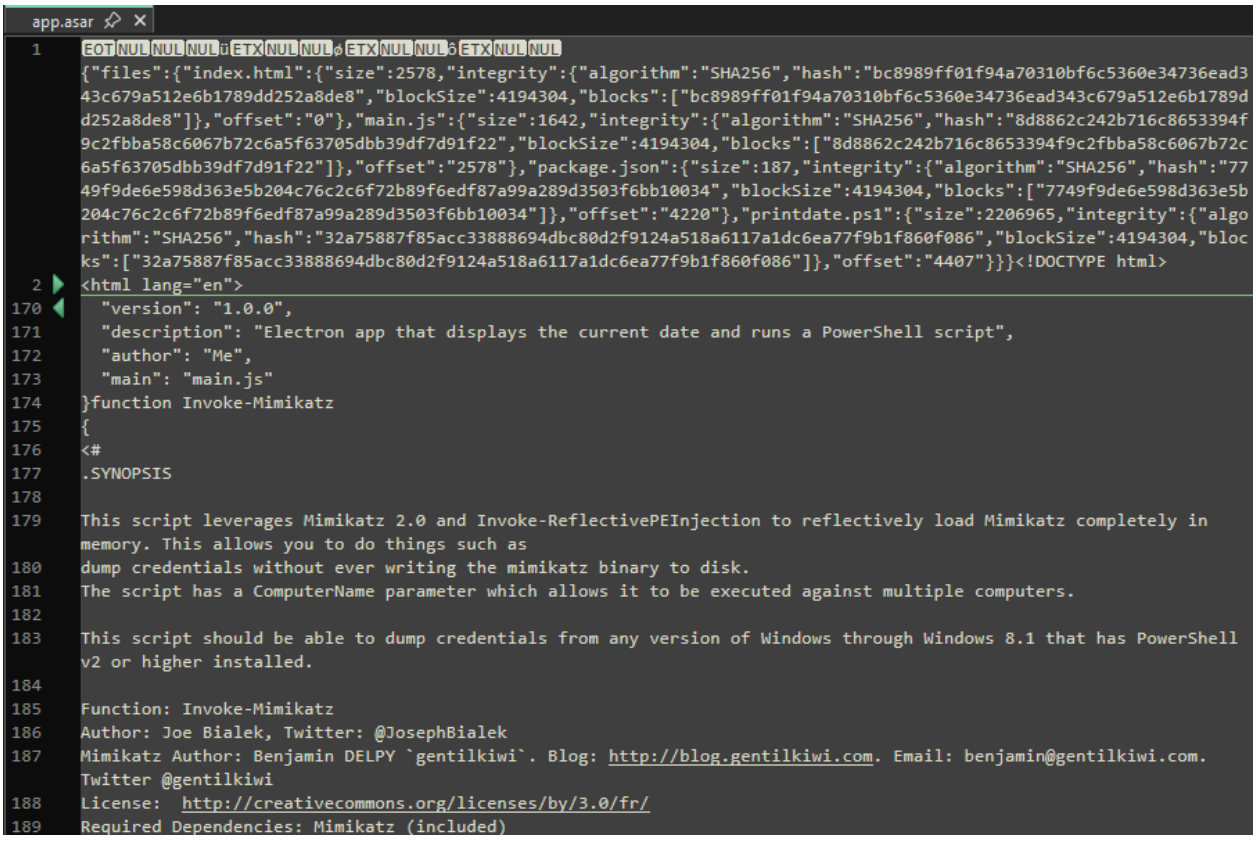
Screenshot of “Date App” application window

Simultaneously, it reads “*printdate.ps1*” from inside the app's asar into memory and creates a Windows named pipe server. Once the pipe is listening, it spawns a new visible PowerShell console (via cmd /c start) whose only command-line argument is a short pipe-connect snippet. As a result, PowerShell connects to the named pipe, reads the full script content into memory via NamedPipeClientStream + StreamReader, then executes it with iex. The pipe server closes after a single connection, and the PowerShell process is detached so the Electron app lifecycle is independent of it. Nothing from “*printdate.ps1*” is ever written to disk.



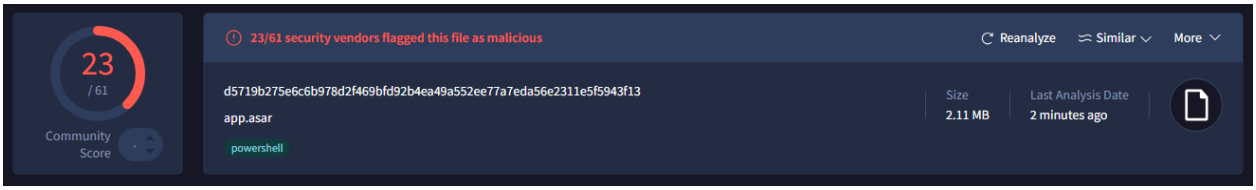
Content of "main.js" file

For the first iteration, inside "*printdate.ps1*" we have put "*Invoke-Mimikatz.ps1*" script from a Powersploit Framework. Content of the script is clearly visible when archived is opened with a text editor application.



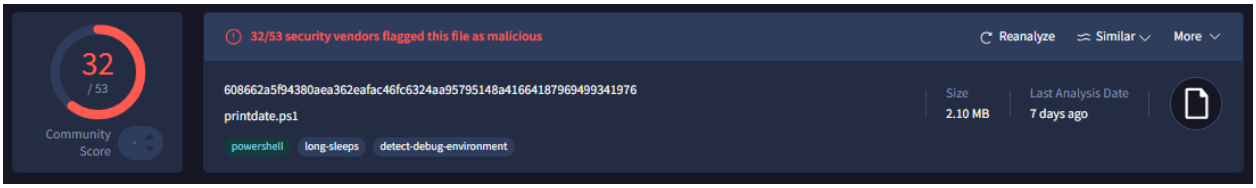
Content of "app.asar" with Invoke-Mimikatz

As in previous test with EICAR, whole "*app.asar*" archive was submitted to VirusTotal and we achieved an identical number of detections.



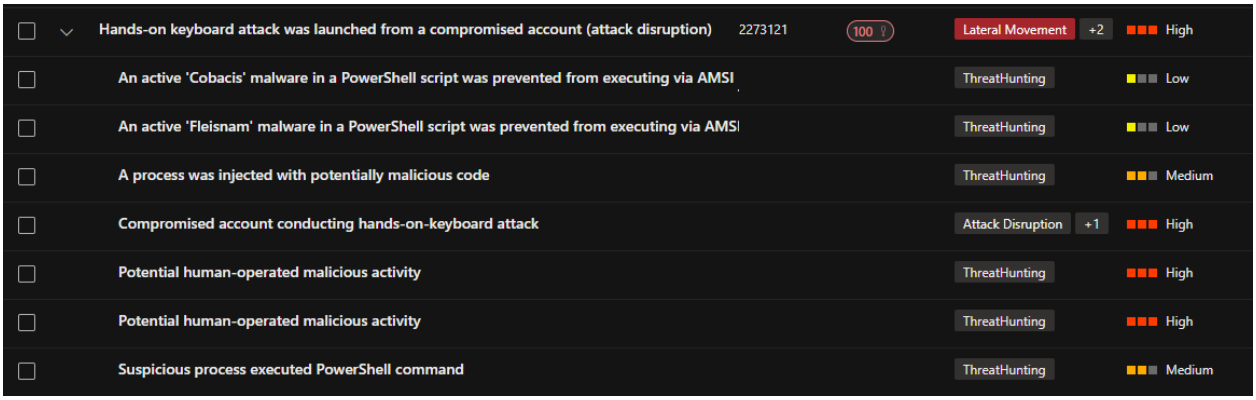
Scanning results of malicious "app.asar" archive from VirusTotal

Difference is not as significant as with the EICAR file, but still there is a significant drop in the number of engines that detected it, when compared with unpacked script.



Scanning results of PowerShell script wit Invoke-Mimikatz from VirusTotal

Application was successfully installed on the Windows 11 Enterprise machine, without triggering Windows Defender. Since we have not obfuscated it in any way, execution of "*Invoke-Mimikatz.ps1*" was stopped and a number of alerts was created on Windows Defender platform.



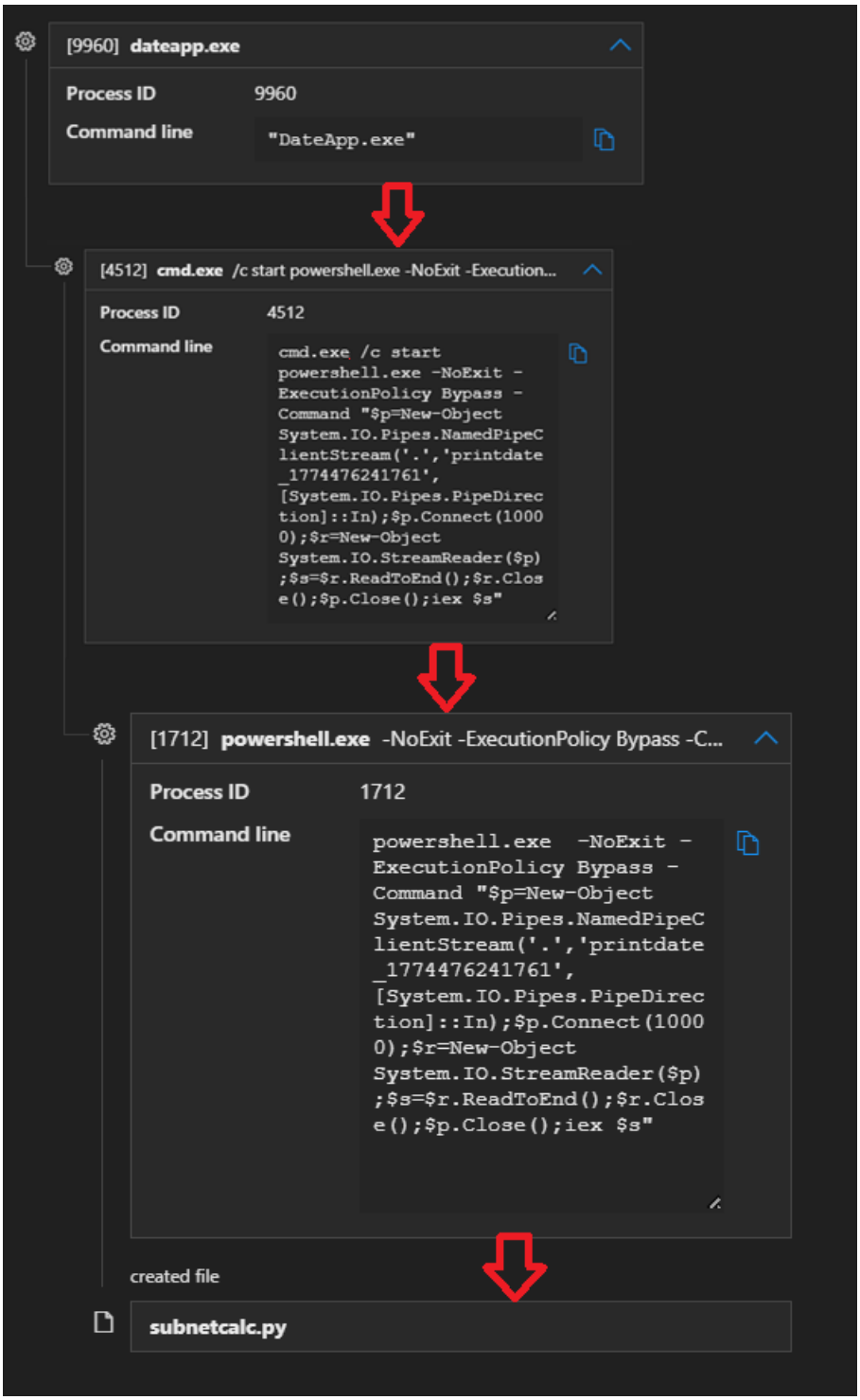
Screenshot of Microsoft Defender for Endpoint detections

Attack was successfully prevented when a well-known malware was used, so for the second iteration "*Invoke-Mimikatz.ps1*" was replaced with simple one-liner containing curl command that downloads a file from GitHub and saves it to "*Public*" directory.

```
curl https://raw.githubusercontent.com/wasmerio/Python-Scripts/refs/heads/main/Subnetting%20Calculator/subnetting_calculator.py -o C:\Users\Public\subnetcalc.py
```

It is important to keep in mind, that binary from which the whole process started stayed the same, as content of "*app.asar*" has no impact on compiled executable.

The second time around our one-liner executed without being stopped or raising any alerts on EDR platform.



Process tree for the creation of subnetcalc.py

There was no “Suspicious process executed PowerShell command” alert, despite it being the exact same Electron application executing the same

PowerShell command as the one recorded when "*Invoke-Mimikatz.ps1*" was used, following the same execution path and using the same main logic. It shows that it was triggered only because a signature-based detection was made and corelated with the previous activity.

10 Difficulty with detecting misuse of ASAR archive

As far as Atos researchers were able to establish based on Windows Defender telemetry, there is only one event that allows for identifying Electron based applications being executed, as there is no metadata that specifically marks binary as coming from Electron framework.
When Electron based application is executed then a "*ProcessCreate*" event is executed with command line directly pointing to "*.asar*" archive containing the application logic.

```
"DateApp.exe" --type=renderer --user-data-dir="C:\Users\<USERNAME>\AppData\Roaming\electron-date-app" --app-path="C:\Program Files\DateApp\resources\app.asar" --no-sandbox --no-zygote --lang=en-US --device-scale-factor=1 --num-raster-threads=4 --enable-main-frame-before-activation --renderer-client-id=4 --time-ticks-at-unix-epoch=-1774429821692381 --launch-time-ticks=1904889953 --mojo-platform-channel-handle=2356 --field-trial-handle=1684,i,18411280016273996454,5502110630849585394,262144 --enable-features=kWebSQLAccess --disable-features=SpareRendererForSitePerProcess,WinDelaySpellcheckServiceInit,WinRetrieveSuggestionsOnlyOnDemand --variations-seed-version /prefetch:1
```

The difficulty of behaviorally detecting misuse of ASAR archives lies in the variety of actions performed by Electron applications. For testing purposes, Atos TRC created a simple correlation rule that identified execution of an Electron application and matched it with executions of Windows Command Prompt, PowerShell, or Microsoft HTML Application Host spawned by the same process within one minute on the same host.
Over a 24-hour period, in a large-scale enterprise organization, this resulted in over 80 different applications and more than 6,000 correlated events.
Command lines varied significantly and showed no clear patterns suitable for whitelisting that could reduce false positives. Some commands could have been interpreted as discovery activity by a threat actor, but in reality they were routine checks performed by the applications themselves.

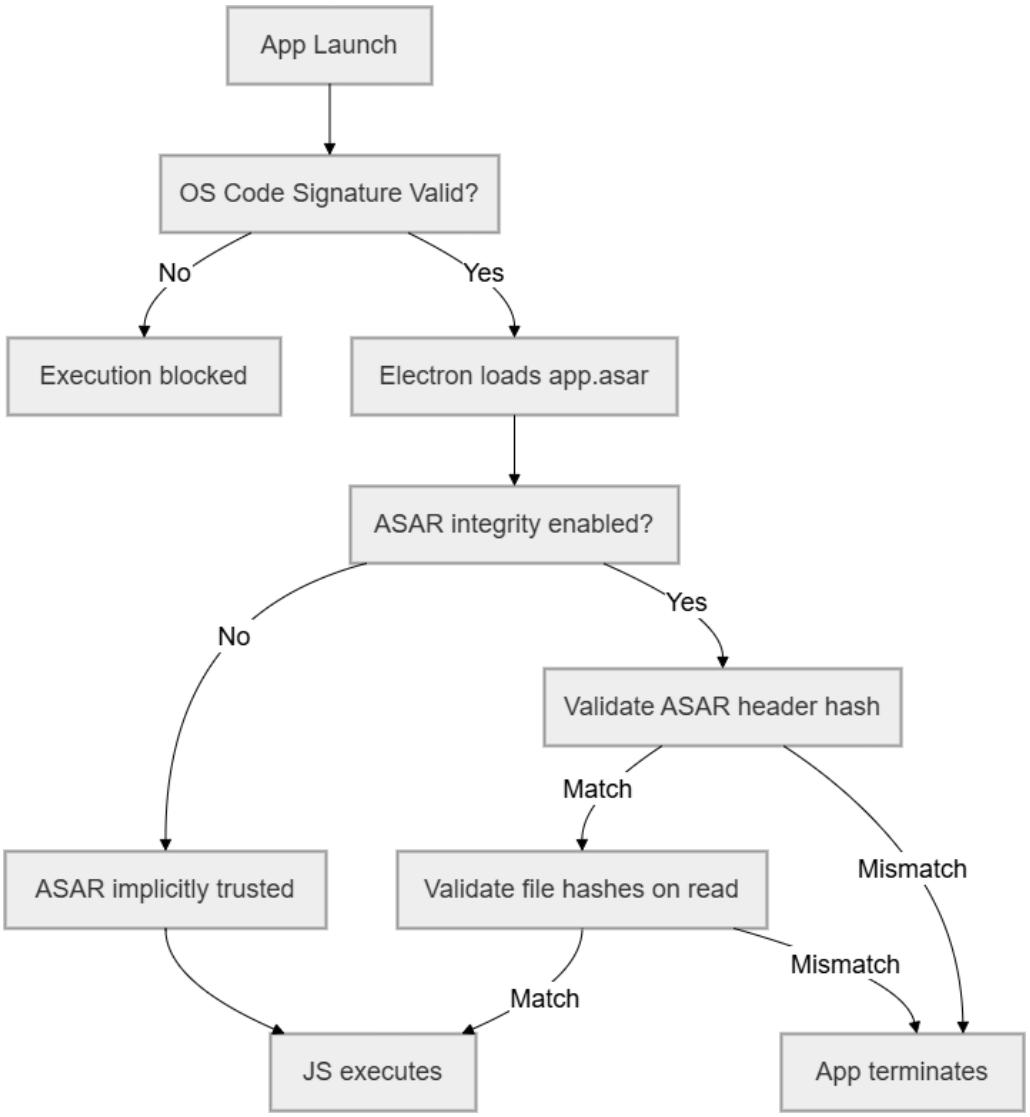
In small, controlled environments, it may be possible to analyze command output and gradually fine-tune detection logic to capture only behavior that deviates from an established baseline. However, in large corporate environments, monitoring dozens - if not hundreds - of Electron applications in this manner is not feasible.
This does not mean that behavioral analysis is not necessary, but it highlights the need for the AV engines to be able to detect malicious code inside ASAR archive prior to its execution.

10.1 ASAR Integrity as additional level of protection

Even though Electron applications don't have antitampering enabled by default, there is a way that developers can protect their applications by enabling ASAR Integrity. According to official Electron page:

> *"ASAR integrity is a security feature that validates the contents of your app's ASAR archives at runtime."*

But how does it work in real life? Enabling ASAR integrity adds cryptographic hashes to the ASAR header so Electron can verify, at file-read time, that the bytes it reads match the metadata describing them, allowing detection of corruption or inconsistencies within the archive. Developers must provide hash of the entire ASAR header when packaging their Electron app and that embedded hash is compared with the one inside ASAR archive. From security perspective this means that application logic cannot be modified without repacking the binary and losing the original certificate.



ASAR Integrity validation logic

```json
{
  "files": {
    "main.js": {
      "offset": "0",
      "size": 12345,
      "integrity": {
        "algorithm": "SHA256",
        "hash": "ab12cd...",
        "blockSize": 4096,
        "blocks": [
          "aa11...",
          "bb22..."
        ]
      }
    }
  }
}
```

Example of integrity metadata from ASAR archive

Despite every integrity check and correct fuses being deployed there are still ways to exploit Electron applications without even touching the ASAR archive, like CVE-2025-55305. Disclosed in September 2025, this critical integrity bypass vulnerability enables attackers with filesystem write access to backdoor trusted Electron applications such as Signal, Slack, and 1Password by subverting internal code integrity checks. Nevertheless, implementing those security measures makes it much less likely for attackers to choose a given application as their attack vector.

11 Key takeaways

ASAR archives are a core component of Electron applications and store application logic in plaintext within a single file that remains executable even when modified. Because ASAR lacks a well-defined and widely recognized magic signature, many antivirus engines treat it as opaque data rather than a structured archive, limiting recursive content inspection and leading to missed detections. This enables attackers to abuse legitimate, signed Electron applications as containers for malicious code by modifying ASAR content without altering the executable itself, a technique repeatedly observed across real-world malware campaigns.

Empirical testing shows a clear and consistent drop in detection rates when malicious scripts are embedded inside ASAR archives compared to standalone files, exposing a structural visibility gap that exists independently of obfuscation or encryption. Although behavioral monitoring can identify suspicious activity originating from Electron applications, reliable detection is hindered by the lack of Electron-specific process metadata and the wide range of legitimate behaviors exhibited by Electron-based software. In large enterprise environments, this results in high false-positive rates and makes scalable, behavior-only detection impractical.

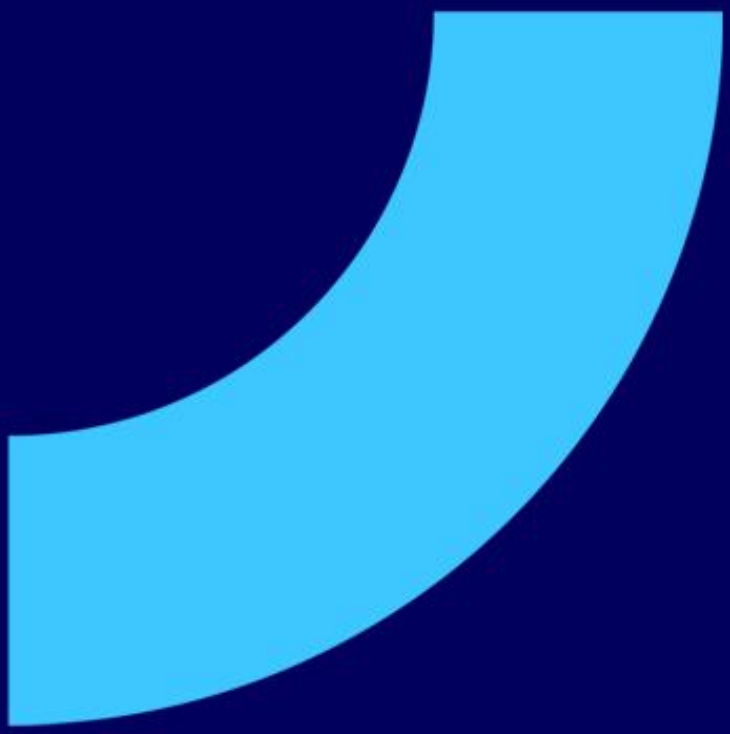
While Electron offers optional ASAR integrity protection, it is not enabled by default, leaving many applications susceptible to silent manipulation of their archived logic. Taken together, these findings highlight ASAR archives as a systemic blind spot across antivirus engines, that should be fixed sooner than later. Reliance on default, signature-driven detections is insufficient to consistently detect or prevent ASAR-based abuse and post-execution behavioral alerts require too much fine-tuning to be an effective tool in large environments, putting therefore much more pressure on the ability of AV engines to be able to properly handle this widely USED archive format.

About Atos Group

Atos Group is a global leader in digital transformation with c. 56,000 employees and annual revenue of c. €7.2 billion (at the go-forward perimeter), operating in 54 countries under two brands - Atos for services and Eviden for products and systems. European number one in cybersecurity and a leader in cloud, Atos Group is committed to a secure and decarbonized future and provides tailored AI-powered, end-to-end solutions for all industries. Atos Group is listed on Euronext Paris.

Find out more about us

atos.net
atos.net/career



Atos