E02. Clipboard-lure deobfuscation analysis

Ilja Krasnjanski
24 October 2026 · Edited evidence report
Historical evidence through 24 July 2026; later engineering checks identified separately.

This report synthesizes the cited case records. Exhibits distinguish retained artifact excerpts from secondary analytical records and claims whose original inputs are unavailable. Excerpts may be redacted or reformatted; extraction methods and limitations are stated locally. Fingerprints identify inputs, not the truth of their claims. Bracketed S labels identify the analysis records listed at the end.

Material and method

The July analysis examined recovered, minified recruitment-page JavaScript with readable strings. The method was static inspection of event handlers, command construction and endpoint references. The sample used in the later companion scanner check is identified below; that scanner check is distinct from the original deobfuscation. [S1, S2]

Clipboard substitution

The recorded copy handler calls `preventDefault()` and writes a generated value through `clipboardData.setData('text/plain', ...)`. In the identified sample the command builder receives a nonce and an operating-system selector; in the `d2saitrade[.]ink` bundle it also receives the phrase, which does not reach the command text. The visible verification instruction therefore differs from the value placed on the clipboard. The handler also records a `cmdline_copied` event. These observations describe browser-side logic, not a captured user session. [S1]

The Unix-like branch constructs a background download-and-execution sequence followed by verification output. The Windows branch prepends hidden installation logic to an interactive access-key prompt. The command still produces a `VERIFY-` result, allowing the visible verification step to accompany the installation attempt. No executable command template is reproduced in this report.

Retrieval and telemetry

The builder derives its download origin from `window.location.origin`. Recorded dropper names include `verify_m` and `verify_w`; the challenge-specific `check_bot` and `check_3d` variants come from other bundles in the lure fleet. Endpoint references across the fleet include device checks, verification, completion and activity tracking; the device-check route appears in the heavier bundles, not in the identified sample. A nonce or phrase returned to the site may support a verification workflow; frontend code alone does not establish how the server validates it or whether payload installation succeeded. [S1]

Interpretation

Clipboard-write calls are useful triage locations, but their presence alone does not prove deception. Here the conclusion depends on comparing the displayed instruction with the generated clipboard value. A later parse-only scanner reported two clipboard-write sites in the held lure sample; it does not independently prove that divergence. Terminal behavior, user execution and backend responses were not reproduced for this report. [S2]

Artifact identity

Lure JavaScript: 73,545 bytes; SHA-256
`5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd`. [S2]

Exhibits

All excerpts come from minified, single-line JavaScript. Unless stated otherwise, each one is hard-wrapped at 94 characters, and a trailing `\` marks every wrap point. Tokens are unchanged. Byte offsets are zero-based. `<ARTIFACT>` in a command means the artifact named in that exhibit. The JavaScript was only read statically and never run.

Exhibit E02-1. Displayed text versus copied value (held lure sample)

Artifact: Recruitment-lure SPA bundle (joincategory[.]xyz build, held reference sample) · 73545 bytes · SHA-256 `5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd`

Location: bytes 45225–46074

Extraction: `dd if=<ARTIFACT> bs=1 skip=45225 count=850`

Supports: copy handler calls `preventDefault()` and writes a generated value via `clipboardData.setData('text/plain', …)`; displayed instruction differs from the clipboard value; `cmdline_copied` event

```
r.textContent=re(v.nonce,_),u.setAttribute(`disabled`,`true`),B(`verify-alert`),J(o,g),J(s,h)}\
;y(),a.addEventListener(`click`,()=>{v=k(),i.value=``,y()}),o.addEventListener(`click`,e=>{e.s\
topPropagation(),q(t.value,o,e=>{z(`verify-alert`,e,`error`)},{copiedClass:g})});let b=async()\
=>{await q(T(v.nonce,_),s,e=>{z(`verify-alert`,e,`error`)},{copiedClass:h})&&await G(`cmdline_\
copied`,{os:_,step:`verify`})};s.addEventListener(`click`,e=>{e.stopPropagation(),b()}),c.addE\
ventListener(`click`,()=>{b()}),c.addEventListener(`keydown`,e=>{(e.key===`Enter`||e.key===` `\
)&&(e.preventDefault(),b())}),c.addEventListener(`copy`,e=>{if(!(window.getSelection()?.toStri\
ng()??``).trim())return;let t=e.target;!(t instanceof Node)||!c.contains(t)||(e.preventDefault\
(),e.clipboardData?.setData(`text/plain`,T(v.nonce,_)),G(`cmdline_copied`,{os:_,step:`verify`}\
))})
```

The command element shows `re(v.nonce,_)` (`r.textContent=…`). Both clipboard paths write `T(v.nonce,_)` instead: the copy-button path through `q()` (which calls `navigator.clipboard.writeText`, offset 43262) and the `copy` event path through `setData`. These two paths are the two clipboard-write sites the parse-only scanner reported. In this sample the identifiers are `c`, `T`, `v` and `G`, not the `R(b.nonce, b.phrase, os)` and `x(...)` shown in the blog post (see E02-7).

Exhibit E02-2. Displayed builder versus copied builder

Artifact: Recruitment-lure SPA bundle (held reference sample) · 73545 bytes · SHA-256 `5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd`

Location: bytes 39435–39642

Extraction: `dd if=<ARTIFACT> bs=1 skip=39435 count=208`

Supports: the Windows branch is always built with the hidden installer; the displayed and copied commands come from different builders

```
var ne={windows:e=>C(e,{includeGuardInstall:!0}),macos:e=>w(e),linux:e=>w(e)};function re(e,t=\
d()){return v(_(t))?te(e):C(e)}function ie(e,t=d()){return ne[_(t)](e)}function T(e,t=d()){ret\
urn y(_(t),ie(e,t))}
```

The displayed builder `re()` returns `te(e)` on Unix-like systems and `C(e)` on Windows. Neither of these contains the installer. The copied builder `T()` goes through table `ne`, which selects `w(e)` on Unix-like systems and `C(e,{includeGuardInstall:!0})` on Windows. Both arguments are a nonce (`e`) and an OS selector (`t`). The phrase is not an argument (see Exhibits E11-3 and E11-6).

Exhibit E02-3. Platform command wrappers (download bodies elided)

Artifact: Recruitment-lure SPA bundle (held reference sample) · 73545 bytes · SHA-256 `5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd`

Location: bytes 38605–39434

Extraction: `dd if=<ARTIFACT> bs=1 skip=38605 count=830`

Supports: Unix-like branch = background download-and-execute followed by `VERIFY-` output; Windows branch = hidden install prepended to an access-key prompt, still yielding `VERIFY-`

```
function b(e){return`…[elided: double-fork background-detach wrapper]… bash -c "${e}"&>/dev/nu\
ll&`}function x(){return b(`…[elided: download-and-execute sequence: fetch ${f()}/${h.unix} to\
 ${g.unix} and run it]…`)}function ee(){return`start -WindowStyle Hidden cmd '…[elided: downlo\
ad-and-execute sequence: fetch ${f()}/${h.windows} to ${g.windows} and run it]…'`}function S()\
{return`read -rp "${p.linux}" KEY;printf "VERIFY-%s\\n" "$(printf "%s|%s" "$KEY" "$NONCE"|base\
64|tr -d "\\n")"`}function C(e,{includeGuardInstall:t=!1}={}){return`powershell -NoProfile -Co\
mmand "$env:NONCE='${e}'; ${t?`${ee()}; `:``}$k=Read-Host '${p.windows}'; $sig=[Convert]::ToBa\
se64String([Text.Encoding]::UTF8.GetBytes($k+'|'+$env:NONCE)); Write-Output ('VERIFY-' + $sig)\
"`}function te(e){return`bash -c 'export NONCE="${e}";${S()}'`}function w(e){return`bash -c 'e\
xport NONCE="${e}";${x()} ${S()}'`}
```

Three template bodies were replaced by `…[elided: …]…` markers so the excerpt cannot be pasted and run. Everything else is verbatim. `w()` (copied, Unix) puts `x()` before the `S()` verification prompt. `te()` (displayed, Unix) contains only `S()`. On Windows, `C()` inserts `ee()` only when `includeGuardInstall` is true.

Exhibit E02-4. Download origin and dropper names

Artifact: Recruitment-lure SPA bundle (held reference sample) · 73545 bytes · SHA-256
5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd

Location: bytes 38149 and 38380

Extraction: `grep -a -o -b -E 'function f\(\)\{[^}]*\}|h=\{windows:verify_w,unix:verify_m\},g=\{[^}]*\}' <ARTIFACT>`

Supports: the builder derives its download origin from `window.location.origin`; dropper names `verify_m` / `verify_w`

```
38149:function f(){return typeof window<`u`&&window.location?.origin?window.location.origin:``}
38380:h={windows:`verify_w`,unix:`verify_m`},g={windows:`verify.cmd`,unix:`verify`}
```

This sample has only the default `verify_m` and `verify_w` names. The challenge-specific variants appear in the d2saitrade[.]link bundle (E02-8).

Exhibit E02-5. Phrase and nonce generation and the client-side `VERIFY-` check

Artifact: Recruitment-lure SPA bundle (held reference sample) · 73545 bytes · SHA-256
5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd

Location: bytes 39643–39852 (phrase pool `E`), then bytes 39853–40243 (functions `D`, `O`, `k`, `A`)

Extraction: `dd if=<ARTIFACT> bs=1 skip=39643 count=210; dd if=<ARTIFACT> bs=1 skip=39853 count=391`

Supports: nonce/phrase verification workflow; limits of what frontend code shows

```
var E=[`WORKSPACE-KEY`,`DEV-SESSION`,`LOCAL-SETUP`,`ENV-CHECK`,`SG-SESSION`,`BUILD-READY`,`STA\
CK-VERIFY`,`RUNTIME-OK`,`TOOLCHAIN-OK`,`SDK-READY`,`NODE-ENV`,`SHELL-READY`,`PATH-VALID`,`CONF\
IG-OK`,`SETUP-TOKEN`];
function D(){let e=crypto.getRandomValues(new Uint8Array(8));return Array.from(e,e=>e.toString\
(16).padStart(2,`0`)).join(``)}function O(){return E[crypto.getRandomValues(new Uint8Array(1))\
[0]%E.length]}function k(){return{phrase:O(),nonce:D()}}function A(e,t,n){let r=e.trim();if(!r\
.startsWith(`VERIFY-`))return!1;try{let[e,i]=atob(r.slice(7)).split(`|`);return e===t&&i===n}c\
atch{return!1}}
```

The page creates the phrase from pool `E` and the nonce as 8 random bytes. It checks a pasted `VERIFY-` value in the browser: `A()` base64-decodes the value and compares `phrase|nonce`. The candidate types the phrase at the terminal prompt. The phrase is not embedded in the copied command. The bundle does not show any server-side check.

Exhibit E02-6. Activity telemetry and route aliases

Artifact: Recruitment-lure SPA bundle (held reference sample) · 73545 bytes · SHA-256
5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd

Location: bytes 42389 (telemetry sender `W`) and 71851 (route table)

Extraction: `grep -a -o -b -E 'async function W\(e,t,n\)\{.{0,420}' <ARTIFACT>; grep -a -o -b -E 'var ke=\[[^]]*\],Ae=\{[^}]*\}' <ARTIFACT>`

Supports: endpoint references to verification, completion and activity tracking

```
42389:async function W(e,t,n){let r=JSON.stringify({type:e,meta:{...t,deviceInfo:n}});try{let \
e=await fetch(`/track-activity`,{method:`POST`,headers:{"Content-Type":`application/json`},bod\
y:r,keepalive:!0});if(!e.ok)throw Error(`track-activity failed (${e.status})`)}catch(t){if(nav\
igator.sendBeacon){navigator.sendBeacon(`/track-activity`,new Blob([r],{type:`application/json\
`}));return}console.warn(`track-activity failed`,e,t instanceof Error?t.me
71851:var ke=[`/`,`/apply`,`/verify`,`/application-received`,`/privacy-policy`],Ae={"/anti-bot\
-check":`/verify`,"/environment-setup":`/verify`,"/complete":`/application-received`}
```

`G()` (E02-1) wraps `W()`, so `cmdline_copied` is posted to `/track-activity`. In this sample, `/anti-bot-check`, `/environment-setup` and `/complete` are only client-side route aliases. The string `/device-check` does not occur in this sample (`grep -c` = 0). It does occur in the d2saitrade[.]link and heavy bundles (E02-9).

Exhibit E02-7. The copy-handler form quoted in the blog post (`c`, `R`, `b`, `x`, `step:'verify'`)

Artifact: Recruitment-lure SPA bundle, d2saitrade[.]link build (site-asset copy; identical to the gunzipped urlscan harvest body) · 101082 bytes · SHA-256 `775b939e40a2aeb56a741b2f4c1f93e77cac59a4484803af6bed25017d70c5c4`

Location: bytes 44826–45041 (builder chain), then bytes 48732–48996 (copy handler)

Extraction: `dd if=<ARTIFACT> bs=1 skip=44826 count=216; dd if=<ARTIFACT> bs=1 skip=48732 count=265`

Supports: the July deobfuscation's copy-handler identifiers; the builder receives nonce, phrase and OS

```
var de={windows:e=>F(e,{includeGuardInstall:!0}),macos:e=>ue(e),linux:e=>ue(e)};function I(e,t\
,n=E()){return oe(A(n))?le(e):F(e)}function L(e,t,n=E()){return de[A(n)](e)}function R(e,t,n=E\
()){return j(A(n),L(e,t,n))}
…[elided: bytes 45042–48731 not shown]…
c.addEventListener(`copy`,e=>{if(!(window.getSelection()?.toString()??``).trim())return;let t=\
e.target;!(t instanceof Node)||!c.contains(t)||(e.preventDefault(),e.clipboardData?.setData(`t\
ext/plain`,R(b.nonce,b.phrase,y)),x(`cmdline_copied`,{os:y,step:`verify`}))})
```

This bundle is the primary source for the identifiers `c`, `R(b.nonce,b.phrase,y)`, `x` and `step:verify` used in the July note and the blog post. `R` accepts the phrase (`t`) and passes it to `L`. Each entry of de

uses only e (the nonce), so the phrase does not reach the command text. A second copy handler, at byte 61839, writes Le(G.phrase,W) with step:design_challenge_render`.

Exhibit E02-8. Challenge-specific dropper names (check_bot_m/w, check_3d_m/w)

Artifact: Recruitment-lure SPA bundle, d2saitrade[.]link build · 101082 bytes · SHA-256 775b939e40a2aeb56a741b2f4c1f93e77cac59a4484803af6bed25017d70c5c4

Location: bytes 50464–50933

Extraction: dd if=<ARTIFACT> bs=1 skip=50464 count=470

Supports: "challenge-specific variants" of the dropper names

```
function we(){return`…[elided: download-and-execute sequence: ${D()}/check_3d_w to upd_w.cmd]…\
`}function Te(){return`…[elided: detach wrapper + download-and-execute sequence: ${D()}/check_\
3d_m to /tmp/upd_m]…`}function Ee(){return`start -WindowStyle Hidden cmd '…[elided: download-a\
nd-execute sequence: ${D()}/check_bot_w to check_bot.cmd]…'`}function De(){return`…[elided: do\
wnload-and-execute sequence: ${D()}/check_bot_m to /tmp/cb]…`}we(),Te();
```

The four template bodies are elided. The dropper path names stay visible. D() in this bundle is the same window.location.origin helper (offset 43399). The /device-check POST is at offset 50111.

Exhibit E02-9. Heavy front-end variant: TensorFlow.js face models and camera before /device-check

Artifact: Lure SPA bundle, dscufo[.]xyz index-CkxWQCI-.js (urlscan-harvested, stored gzip-compressed) · 295886 bytes (1076463 bytes decompressed) · SHA-256 63b47a480acca87c685e48bb9e9e0306313b322e724af85fa5a86db014232114 (decompressed: 69cf5e6cc55ccb3ec8a9b43841cad61c7a3832b4e3114a5ff51151bd02ab8ae6)

Location: decompressed stream, bytes 1005815–1006275, then bytes 743440–743669

Extraction: zcat <ARTIFACT> | dd bs=1 skip=1005815 count=461; zcat <ARTIFACT> | dd bs=1 skip=743440 count=230

Supports (blog post 01): the larger bundles pull in TensorFlow.js for a device-check presentation

```
y=K.useCallback(async()=>{const R="/models";await Promise.all([Xn.tinyFaceDetector.loadFromUri\
(R),Xn.faceLandmark68Net.loadFromUri(R)]);const N=await navigator.mediaDevices.getUserMedia({v\
ideo:!0});o.current=N,r.current&&(r.current.srcObject=N,r.current.onloadedmetadata=()=>{r.curr\
ent.play(),m(!0)})},[]),x=K.useCallback(async()=>{l("checking");try{(await st.post("/device-ch\
eck",{company:location.href})).data?.result?l("ready"):l("blocked")}catch(R){console.
…
nd, which binds to TensorFlow C++, by running npm i @tensorflow/tfjs-node, or npm i @tensorflo\
w/tfjs-node-gpu if you have CUDA. Then call require('@tensorflow/tfjs-node'); (-gpu suffix for\
 CUDA) at the start of your program. Visit
```

The face-detector models load from /models and the camera starts before the /device-check POST. The second span is the TensorFlow.js runtime's own backend message, bundled in. The other three heavy bodies, dscufo H6EfDx2S and nemesistrade[.]work CgHQI6a2/TCXLGCej, give the same counts: tinyFaceDetector=3, faceLandmark68Net=3 and tfjs=8 in each body. None of the four heavy bodies contains three.js markers (WebGLRenderer, BufferGeometry, Object3D and PerspectiveCamera are all 0).

Exhibit E02-10. Where three.js actually appears: the SceneBox design-challenge lure

Artifact: SceneBox lure front-end script (unminified) · 22012 bytes · SHA-256 ce7865ba3b511bba88c428439651c464d7f66a5a01b4c814883b62c9c6cc32fa

Location: lines 1–2 and 384

Extraction: sed -n '1,2p;384p' <ARTIFACT>

Supports (blog post 01): the three.js half of the "TensorFlow.js and three.js" statement

```
import * as THREE from "three";
import { OrbitControls } from "three/addons/controls/OrbitControls.js";
const renderer = new THREE.WebGLRenderer({ antialias: true, alpha: false });
```

three.js is present here, in a separate 22 KB 3D-preview lure that sits next to retained check_3d_m/check_3d_w files. It is not in the TensorFlow.js bundles of E02-9. This file contains no TensorFlow.js (tensorflow = 0).

Claims without a retained primary artifact

- The companion's parse-only scanner result ("two clipboard-write sites") comes only from the validation record. The sample itself does have two write paths (E02-1), but the scanner output is not a primary artifact.
- Terminal behaviour, user execution and backend responses were not reproduced, and no artifact exists for them (as the report already states).

Exhibit inputs

Retained inputs identified by size and SHA-256. These include artifacts and analytical records; each exhibit identifies its evidence type and extraction limitations. The files themselves are not distributed.

Recruitment-lure SPA bundle (held reference sample) · 73,545 bytes

`SHA-256 5ee915bb8d0363bd92176cc595d7425fd7e7830508d8824d90b5285a3d2b11bd`

Recruitment-lure SPA bundle, d2saitrade[.]link build · 101,082 bytes

`SHA-256 775b939e40a2aeb56a741b2f4c1f93e77cac59a4484803af6bed25017d70c5c4`

SceneBox lure front-end script (unminified) · 22,012 bytes

`SHA-256 ce7865ba3b511bba88c428439651c464d7f66a5a01b4c814883b62c9c6cc32fa`

Source records and fingerprints

SHA-256 identifies the reviewed source bytes, not acquisition time or every artifact described. Restricted originals are not reproduced.

[S1] Clipboard-handler analysis, 23 July 2026 · 4,662 bytes

`SHA-256 2cdf96b2a4ef16fdaf987ae73260e3531564e599aebadfad505a4c2b65a19a47`

[S2] Companion validation record, 21 October 2026 · 2,562 bytes

`SHA-256 613d16e0bed012c96e00a7db6e2f7260b263fa5cd3941d39d523a71dfc94a74d`