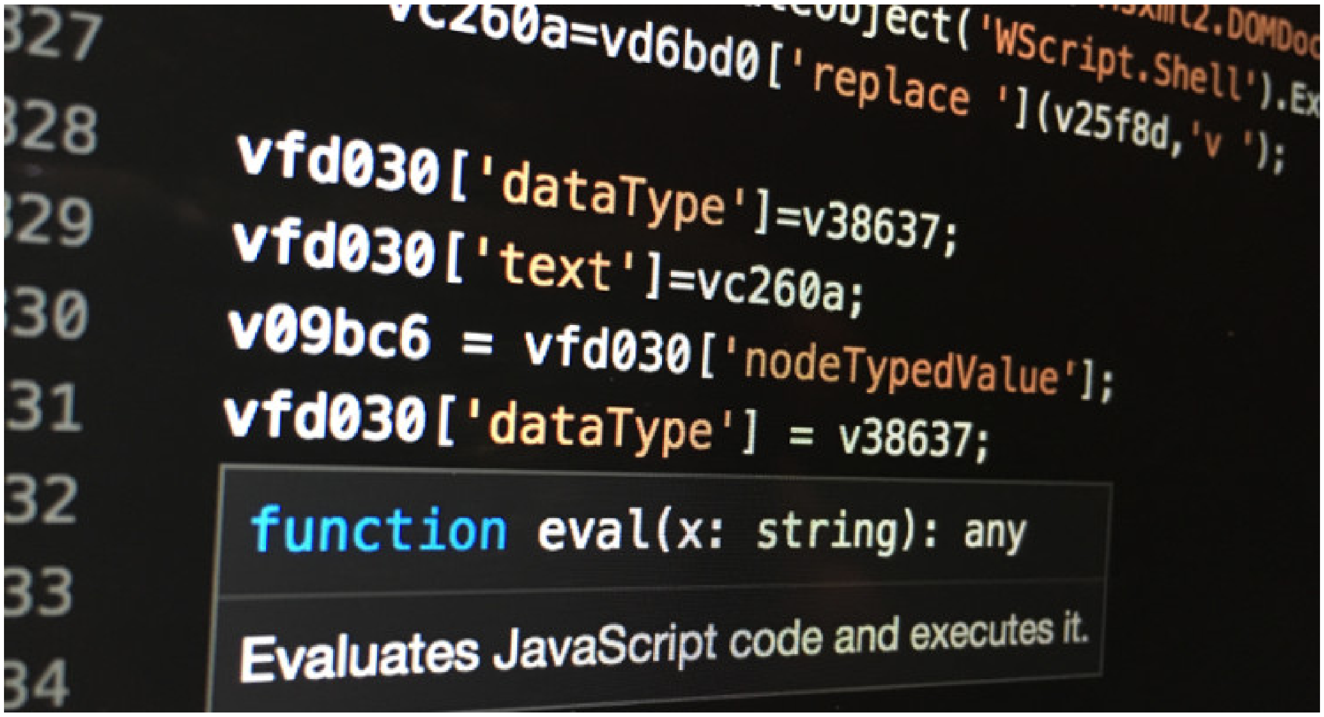
Spora Ransomware: Understanding the HTA Infection Vector

linkedin.com/pulse/spora-ransomware-understanding-hta-infection-vector-kevin-douglas

Kevin Douglas



```
                                   ject('WScript.Shell').Ex
  c260a=vd6bd0['replace '](v25f8d,'v ');
  vfd030['dataType']=v38637;
  vfd030['text']=vc260a;
  v09bc6 = vfd030['nodeTypedValue'];
  vfd030['dataType'] = v38637;
```

function eval(x: string): any

Evaluates JavaScript code and executes it.

Kevin Douglas

Kevin Douglas

Technical Director at Tenable

Published Mar 9, 2017

+ Follow

ABSTRACT

Recently, the MalwareHunterTeam announced the discovery of new ransomware. This ransomware, Spora, is one of the most sophisticated examples seen to date. According to BleepingComputer, Spora has "*top notch encryption*", "*has the most sophisticated payment site as of yet*" and a "*professional decryption service*". Even its infection vector demonstrates a level of sophistication above and beyond its predecessors.

One of Spora's infection vectors is via an HTA email attachment. During the infection, no additional network communication is needed – this malware dropper is completely self-contained. The HTA contains an embedded copy of Spora that is decrypted and installed on

the victim machine as the attachment is opened. If you have the attachment, you already have Spora – network connection, or not.

This is the perfect opportunity for us to learn about HTA-based malware droppers. Lets take a look at the Spora infection vector to learn why the authors chose HTA…

BACKGROUND

Most of us are already familiar with typical infection vectors from droppers that leverage Microsoft Office files, PDF documents, Flash files, etc. Spora has chosen to use HTA as the malware dropper. Interesting choice… So, what is HTA?

HTA stands for HTML Application. It is a technology that Microsoft created to allow graphical user interfaces to be bundled directly with script logic. Scripting languages such as VBScript do not have any native GUI capabilities. HTA leverages Internet Explorer to provide scripts with a graphical interface. Even though they leverage Internet Explorer, they run outside of the browser via MSHTA.exe. This means that they are free from the normal security restrictions placed on scripts that run inside a web browser.

To illustrate, an existing HTML file can be renamed to an .hta extension and it would yield a working example of HTA. The only difference being, the document would open via MSHTA.exe rather than a web browser.

> “ *HTAs, by contrast, are not bound by the same security restrictions as Internet Explorer; that’s because HTAs run in a different process than Internet Explorer. (HTAs run in the Mshta.exe process rather than the Iexplore.exe process.) Unlike HTML pages, HTAs can run client-side scripts and they have access to the file system. Among other things, this means that HTAs can run your system administration scripts, including those that use WMI and ADSI. Your scripts will run just fine, and you won’t receive any warnings about items that might be unsafe.*” – Microsoft TechNet

For a malware author targeting Microsoft Windows victims, HTA seems like a reasonable choice. Its simple, it can (“*among other things*”) access the file system, and it is a pre-installed capability in Windows.

Now, lets take a deep dive into how the authors of Spora leveraged HTA for their infection…

ANALYZING THE INFECTION VECTOR

OVERVIEW

The Spora infection begins as an HTA file.

Like many other malware droppers, it is reported that the Spora HTA file is initially delivered to its target via an email attachment. The delivery mechanism doesn't matter so much, as long as the HTA file reaches a victim and the victim is likely to click on it.

Spora disguises the HTA file name such that it includes two file extensions (e.g., <*filename*>.doc.hta). This is most likely an attempt by the malware author to hide the actual .hta file extension from the victim. On Windows machines that are configured to hide file extensions, the actual file extension (.hta) would be hidden from the user, leaving only the fake file extension displayed (e.g., <*filename*>.doc appears rather than <*filename*>.doc.hta). The intent is to deceive users into clicking on the file, thinking that they are opening a Word document rather than the HTA file.

Once the HTA file is opened, the infection occurs through a series of five distinct stages as shown below. In stage one, the HTA dropper writes an embedded payload to a JScript file **(%temp%/close.js**) and runs it. This JScript file contains the entire payload necessary for each of the subsequent stages.

Stages two, three and four each decode an embedded JavaScript payload and run it. Stage two is distinct in that it relies heavily on anonymous self-invoking functions as a form of obfuscation. Stage three is distinct in that it uses a custom decode logic to decode its payload prior to running it. Stage four is the most complex, leveraging AES encryption via CryptoJS to decode the final payload to run. Stage five runs the final JavaScript payload in order to create a Word Document (***doc_6d518e.docx***) and install/drop the Spora malware (***81063163ded.exe***). Stage five then opens the Word document and starts the Spora malware.

The Word document serves no purpose other than to distract the victim from noticing that Spora was installed. When the victim sees a Word document opened from an email attachment with a .doc extension, he/she is more likely to think that nothing is wrong. Actually though… the infection is done. Spora is now encrypting precious files on the victim's hard drive and holding them for ransom.

Lets walk through this logic in detail to see how each of these stages work…

Stage 1: Create close.js and Run It

In the first stage of the infection, Spora uses an HTA file as a container for each component used during the infection process. As shown below, this infected HTA file is a simple file with a single purpose - to write its embedded payload to a JavaScript file and run it.

In the first step of this stage, a JavaScript file (***close.js***) is created in the ***%TEMP%*** folder. As shown below, ***WScript.Shell*** and ***Scripting.FileSystemObject*** objects as are used to create this file. Leveraging these components is a fairly common technique used by malware droppers to gain read/write access to the local file system.

The next step is to write the infection payload to this file. As highlighted below, a payload is written to the file using the ***Scripting.FileSystemObject*** object. The highlighted payload was abbreviated in the screen shot for conciseness and readability. The actual payload is much larger than what is shown.

Once the payload has been written to the file, the file is closed and then run using the ***WScript.Shell*** component. Highlighted below, we can see that ***%temp%\close.js*** is being executed.

The first stage of the infection is now completed. As the victim opened the HTA email attachment, the embedded JavaScript payload has been extracted from the HTA file, written to a JScript dropper and started.

In stage two, lets see what was in the payload and step through what it does…

Stage 2: Decode JavaScript Payload and Run It

The screen shot below shows the partial content of the payload extracted during stage one and written to ***close.js.*** The beginning of the file appears to contain JavaScript. However, much of the file appears to be something other than JavaScript. Our hunch is that this is an additional payload. Most likely, the JavaScript that we see is merely a thin wrapper that decodes the payload and launches it.

This file is obviously formatted as ugly as possible by the author in an attempt to make it less readable. Reading and understanding this file in its current form is a challenge. We will need to reformat it (add white-spacing, etc.) in order to be able to make any sense of it.

After beautifying the code as shown below, we can start to see two distinct parts to the file. The first part is obviously JavaScript. Most likely the ***Gr()*** function starting on line 5 is decode logic. The second part, starting at line 28, appears to be payloads that are passed into the ***Gr()*** and ***qP()*** functions.

So, lets abbreviate some of the payloads to allow us to get a better overall picture of the logic without being confused by the payload values themselves. Shown below, we can start to see that many payloads are passed into the suspected ***Gr()*** decode function.

Looking at the definition of ***qP*** highlighted below, we see that it is simply assigned to the JavaScript ***Function*** constructor. This is simply a form of obfuscation where the malware author is trying to hide each time ***Function*** is called by reassigning it to the ***qP*** variable. In other words, if the code invokes ***qP()***, the net effect is that ***Function()*** is called. We will see this in practice shortly…

Based on our discovery above, lets replace all ***qP()*** references with ***Function()*** to make the code more readable. Shown below is the resulting cleansed code.

So, is the ***Gr()*** function really a decode loop?

Looking at the highlighted logic shown below, we can see that the variable ***x*** that is passed into ***Gr()***, is iterated over, its values are altered using some basic math formulas resulting in a new value (variable ***b***). The new array of values is joined back together into a string and returned. Long story short, this is definitely a decode loop.

Based on our analysis above, lets rename all the ***Gr()*** references to ***Decode()*** to make things more readable as shown below.

The next thing to decipher is the **(function() { … })()** syntax highlighted in the abbreviated screen shot below. This syntax indicates a JavaScript anonymous self-invoking function. The function is anonymous because it has no name specified. The function is self-invoking because it is wrapped in parenthesis and followed by a pair of parenthesis. Long story short, this syntax makes it an anonymous self-invoking function, meaning it will automatically invoke at startup without explicitly being called.

Now that we know how to identify an anonymous self-invoking function, we can easily spot a second one as highlighted below. As ***close.js*** is started, the outermost self-invoking function shown above will automatically run. As it runs, this innermost self-invoking function shown below will automatically start.

As we look at the function shown above, we can see a difference between the innermost and the outermost function syntax. The innermost function highlighted above is capitalized. Why? This is an example of invoking a function constructor. The JavaScript function constructor allows a function to be defined and invoked in the same line of code.

The syntax for defining and invoking a function constructor is ***Function(arg1, arg2, argn, codeblock)***. The last argument passed into the Function constructor is the code block for the function itself. This means that the highlighted section of code below is the code block for this innermost self-invoking function on line 28.

So far, we know that as ***close.js*** starts, it self-invokes the outermost function on line 2. It then self-invokes the innermost function on line 28. The innermost function is a function constructor, which means that the highlighted section of code is ultimately the code that automatically runs at startup time. After the highlighted section of code finishes running, its output becomes the code block argument to the innermost function as it starts.

We're getting closer…

Looking at the code highlighted above, we can see, similar to line 28, it is a function constructor. This means that the highlighted section of code below represents its code block (since it is the last argument passed into the function constructor).

If the above highlighted line of code is the code block for the function constructor, then what is this similar looking section of code highlighted below? These are the arguments passed into the code block highlighted above. If we decode the code block above, we should see logic that references the arguments highlighted below.

Time to decode the code block and see what this stage of infection actually does…

If we use the ***Decode()/Gr()*** logic shown earlier to decode the first payload, we can see the code block as shown below. As we suspected, we can see references to inbound arguments in lines 11 and 12. The logic appears to iterate over each of the bytes in each of the arguments, alter them with some basic math formulas and join the results back together.

Sound familiar? Yes, it's another decode loop.

Based on our understanding of the logic above, it appears to decode each argument passed into it. Looking at the highlighted line below, we can see there are twenty-one arguments passed into this decode loop. These arguments are first decoded via the ***Decode()/Gr()*** function discussed earlier and then further decoded with this new decode loop shown above.

Ultimately, this dual-decoded array of payloads shown below is the infection code that is run at the end of this stage.

This stage of the infection began with a series of nested self-invoking functions, which automatically ran at startup time (innermost first, followed by the outermost). The innermost self-invoking function dynamically built and ran a function constructor. The code block for this function constructor was the result of decoding an embedded payload from ***close.js***. Ironically, this code block turned out to be second decoding loop.

As this function constructor ran, the second decode loop was passed twenty-one decoded payload chunks from ***close.js*** as arguments. The second decode loop decoded them once more, resulting in the final payload to be run by the innermost self-invoking function.

We are getting closer. Lets take a look at what this new JavaScript payload does…

Stage 3: Decode JavaScript Payload and Run It

It looks like our analysis was correct. The resulting payload that was run at the end of stage two shown below is obviously JavaScript. The good news is that this stage appears to be fairly simple. There appears to be a payload, a decode loop and an ***eval()*** statement to launch the payload once it has been decoded.

The highlighted section shown below (abbreviated in the display for readability) appears to be a payload along with some JavaScript function names (e.g., ***fromCharCode***). Our hunch is that this payload will be decoded. The decoder will most likely reference the JavaScript functions in lines 4-6 in order to obfuscate the logic and avoid detection.

So, why do we think this is a decode loop?

The first hint is that the ***eval()*** statement at line 39 launches the results from our suspected decode function ***0x1c508().*** This means that whatever is returned from ***0x1c508()*** must be runnable JavaScript. Otherwise, the ***eval()*** function would be useless.

The second hint is that the ***0x1c508()*** function is littered with references to our payload variable ***_0x1C4F5[]***. It references both the suspected JavaScript payload section of ***_0x1C4F5[]*** as well as the other artifacts (e.g., JavaScript function names). This is more evidence that this function most likely decodes a portion of the payload and uses the JavaScript references in the payload as part of this decode logic.

The third hint is that line 36 returns the variable ***_0x1C554***. This variable is built inside a loop in lines 31-34. If the variable is built inside a loop, returned as the result of this function, and has to be JavaScript (since the return value is passed into ***eval()***), then this is definitely a decode loop.

As compared to stage two, this stage of the infection was relatively simple. This stage contained a payload, which was comprised of encoded JavaScript and JavaScript function names. The decode logic was obfuscated through the use of these JavaScript function names rather than invoking them directly. Once decoded, the payload was launched via the ***eval()*** function.

Its time to decode this payload to see if our analysis is correct so far…

Stage 4: Decode Final Payload and Run It

The payload decoded in stage 3 is definitely JavaScript as shown in the output below.

Similar to the previous stage, we can see an obvious payload at lines 2-3 and an ***eval()*** function at line 40 which will be used to run the payload. However, this time we don't see a decode loop. We do notice several references on lines 9 and 16 to CryptoJS, which is a cryptography package for JavaScript. Could CryptoJS be used to decode the payload rather than our previously seen decode loop techniques?

Lets dig deeper to find out…

Our suspected payload is highlighted in the screen shot below. This payload was abbreviated for conciseness and readability.

Expanding the payload display as shown below gives us a better idea of the size of the payload contained in this infection stage. The payload is actually an array of hex encoded values (many separate payloads in a comma separated list). Since it is an array of payloads, each payload element in the list may be used for different purposes at different times during this infection stage.

Please note that this is still only a representative clip of the overall payload.

Looking at the highlighted section of the payload array shown above, the hexadecimal values fall within the printable character set range (e.g., 0x20 – 0x7E). We can easily convert them from hexadecimal to ASCII to gain a better understanding of the payload. Maybe its simply hexadecimal encoded JavaScript?

Converting portions of the payload array yields some interesting results as shown below. Its not JavaScript, but, its definitely interesting.

Looking at the results above, we can see references to cryptography (e.g., AES, BlockCipher, encryptBlock, decryptBlock, OpenSSL, etc.). We can see references to ADODB.stream, which is typically used by droppers to write malware to disk. We can see references to WScript.shell, which is typically used by droppers to run malware after its been installed on disk. We can also see references to what appears to be a Word document name (e.g., ***MP%\doc_6d518e.docx***).

Using the list of decoded hexadecimal values above, we can layer the actual values into the original code as shown below to gain more insight into the intent of this infection stage. In the code below, references to the original payload (e.g., ***_0xee6f[]***) were replaced with their hexadecimal decoded values. In some cases, lines of code that were performing simple string concatenations were collapsed together to simplify the code shown.

We can now see a clearer picture of the infection. Analyzing the code, we see an obvious payload that appears to be decrypted via CryptoJS using AES in line 7 below. We can also see the crypto key as shown (***1C1614D7***). Further analyzing the code, we can now see references in lines 13 and 14 to a Word Document ***(%TEMP%\doc_6d518e.docx***) and an EXE file ***(%TEMP%\81063163ded.exe***).

We are getting **extremely** close to the final stage of the infection. Lets push onward…

The payload shown above is UTF8 encoded and stored in the variable ***v998c9*** as shown in line 15 below.

This AES decrypted payload is then run in line 40 shown below using the ***eval()*** function. This is the payload we need to see decoded. It appears to be the final stage of the infection.

In this stage, we see a payload with sections of it that decode to reveal references to AES cryptography. We also see signs of well-known techniques used by droppers to write to disk and run/start files. We see references to a Word document and we see a reference to an EXE. We see a payload that is AES decrypted and ultimately run via an ***eval()*** function.

Lets decode the payload that is run in line 40 above and reveal the mystery…

Stage 5: Spora infection completes

The full decode of the JavaScript payload from stage four is shown below. There is a big difference between the JavaScript in this stage and the previous stages. Each of the previous stages were heavily obfuscated and focused on decoding payloads and running them. This code is straightforward. It opens two files, writes data to each one, saves them to disk, closes them and then runs them. Our hunch is that one of these files is the Word document and the other is the EXE referenced in stage four.

Lets take a look and see if we are correct…

In line 4 shown below, the contents of the variable ***v09bc6*** are written to disk. If we reference the cleansed code from stage four, we can see that ***v09bc6*** contains the data bytes for the EXE file ***(%TEMP%\81063163ded.exe***).

The ***v3462e*** variable is pointing to a reference of ***Adodb.Stream***. This means that the highlighted line below is equivalent to ***Adodb.Stream.Write(v09cb6)***. As we suspected in stage four, ***Adodb.Stream*** is used by this infection to access and write to the local file system.

Once the data has been written, the file is saved to disk as shown below. Referencing the cleansed code in stage four, we can see that the ***v759c1*** variable is set to ***%TEMP%\81063163ded.exe*** in line 14 of the code. As the result, this line of code is used to create an EXE on the local file system.

This is the code that actually dropped the Spora malware to disk!! Without this line of code, no infection would occur.

Now that Spora exists on disk, it needs to be started. The code shown below is the code that actually runs Spora.

The ***vefa30*** variable is set to the same value as ***v759c1*** in line 16 in the cleansed code from stage four. Both variables point to the file name ***%TEMP%\81063163ded.exe***. It may be confusing that the variables are different in the above and below code fragments. Most likely, it is either due to poor programming style, or, intentional obfuscation.

Similar to the steps above, another file is created as shown below. This time, it is the Word document being created and saved.

Shown below is where the Word document is actually saved to disk. Referencing the cleanse code from stage four, we can see that the ***v365ec*** variable is set to ***%TEMP%\doc_6d518e.docx*** in line 13 of the code.

The Word document is opened/run via the highlighted code shown below. Unlike the EXE example above, the same variable ***v365ec*** was used in the ***SaveToFile*** and the ***Run*** logic.

CONCLUSION

The Spora infection vector using an HTA-based email attachment was more sophisticated and complex than what is seen in the typical malware dropper. Typically, a dropper may leverage several stages of payload decodes prior to ultimately dropping and running the malware. Each of these stages would typically rely on simple hex encoding of the payload combined with some lightweight custom decoding logic.

Spora took infection vector obfuscation to a new level. Rather than one or two stages, Spora used five distinct stages to infect the victim. These stages leveraged hex encoded payloads, custom encoded payloads and AES encrypted payloads. The infection vector transitioned from its initial HTA-based VBScript file to creating and running a Jscript file to complete the infection. The decode loops themselves were obfuscated to leverage JavaScript function names pulled directly from the payload rather than naming them directly. The dropper code leveraged esoteric capabilities of JavaScript, such as anonymous self-invoking functions, to obfuscate its intentions. Finally, the dropper was self-contained to include the Spora EXE as well as a dummy Word Document to confuse the victim. No additional network connectivity was needed once the victim received the dropper.

Other droppers have used many of these techniques in the past. However, the combination of these techniques, combined with the degree to which the Spora dropper leveraged them, definitely set a new standard of sophistication and obfuscation.

Thanks for reading...

You can also connect with me on Twitter at @kd_cybersec.

31 4 Comments
Like Comment Share

➜ Peter Berson
I feel that all new malware is about obfuscation and levels of encryption in the payloads to circumvent most endpoint A/V engines. This is why a lot of them are trying sandboxing approaches. Machine Learning and Deep Learning will help in the next gen A/V endpoint clients to detect this better. The actual ransomware and command and control part is simple, it's more about getting the payloads to execute and evade A/V protection at all levels. Firewalls, Anti-Spam systems, and endpoint. Great job and explaining multi-stage approach.

Sign in to like this comment

Sign in to reply to this comment

1 Like
5y

Roger Cruz
Great explanation. Thank you for taking the time to teach us these internal details. I learned quite a bit
Sign in to like this comment

Sign in to reply to this comment

1 Like
5y

Evgueni Loukipoudis
Z
Sign in to like this comment

Sign in to reply to this comment

5y

Tim Wiser 🇺🇦
Superb breakdown of infection. Fascinating read !!
Sign in to like this comment

Sign in to reply to this comment

1 Like
5y
See more comments
To view or add a comment, sign in