Stages:

Static properties analysis

- PeStudio
  - Indicators section very useful :potentially malicious, very useful
  - Hashes :useful to look up hashes online
  - Strings :look for interesting
  - Import :look for common malware library imports
  - Sections :i.e. .exe might consist of .text .data .rsrc .reloc, ioc dev
  - In dropper example we can extract with PeStudio resources, dump raw.
- Strings (look for stuff like network iocs/ tmp/config files /reg keys / communication mechanisms)
  - pestr malware.exe | more :look for interesting strings
  - pestr --net :extracts strings only looking like network related details
  - strings -a :search twice, first for ASCII (-a searches whole file)
  - strings -a --encoding=l :second search for Unicode strings
- CFF Explorer
- Peframe
  - Peframe malware.exe | more
    - Hashes
    - Detected (Packer & type / possible anti debug) :flags GetLastError but ?
    - Suspicious APIs
    - Anti Debug APIs
- Detect It Easy (diec is cmd line)
  - PE header details (especially useful for determining which tools used to generate exe)
  - Packers
- Exeinfo
  - *Exeinfo has capability to rip/carve files the file embedded artifacts
  - PE header details (especially useful for determining which tools used to generate exe)
  - Packers used
- Signsrch
  - Locates code used for rypto compression more
- PeScan/PortEx
  - Examine key aspects of Windows exe files and id some anomalies
- MASTIFF
  - Extracts many details from va types of malware; good for bulk review of many samples
- Exiftool
  - Displays metadata embedded in files of various types
- TrID
  - Examines type of file you're starting to examine
- Viper
  - Manages malware collection and extractions various properties about files
- reg_export: sometimes regedit can't properly export, use reg_export

Interactive behavior analysis

- Process Hacker
  - Launch first, baseline process listing, svc, network connections

    - If ASLR, disable
- Process Monitor
    - Launch after Proces Hacker, clear log, activate immediately before infection
    - Records interactions w/registry, file system, network, other processes
    - During analysis can filter on Process Name (Cntrl+F/is malware.exe)
    - Example look for WriteFile operations to tmp files, can help
- ProcDOT
    - Visualizes Process Monitor logs for easier analysis & helps filter noise
    - Load Process Monitor logs CSV
    - Can import Wireshark pcap as well
- RegShot
    - Click 1st shot button, then click the shot button (baseline)
    - Highlights changes to file system and the registry
    - After giving malware enough time, click 2nd shot button, click shot, then compare
- Wireshark (use w/TCPLogView)
    - Good backup to other tools to capture *all* network traffic
    - Can use with *httpd start* & fakedns to look for GET type requests
- Fiddler
    - intercepter for HTTP(S); proxy like debugging
    - Can use AutoResponder in which you don't have active listeners
- TCPLogView
    - helps show what local process involved with network connections
- Fakedns
    - Help keep isolated
    - *Note for hard coded ips to redirect, -A enables & -D disables
        - iptables -t nat -A PREROUTING -I eth0 -j REDIRECT
        - accept-all-ips start/stop; REMnux cmd to do iptables redirect
- INetSim
    - Emulate various protocols like HTTP(S), SMTP, FTP, POP3, TFTP, IRC, etc
    - /var/log/inetsim contains log files, most useful is service.log contains details about requests INetSim recd & responses
- TCPLogView
    - Maintains a historical log of local TCP connections & corresponding processes
- PolarProxy
    - Intecepts & decrypts SSL/TLS traffic in lab
- ApateDNS
    - DNS server for redir hostname resolution req; similar to fakedns but runs on windows
- FakeNet-NG
    - Intercepts network traffic in lab emulating common protocols; similar to INetSim but runs on Windows
- scdbg
    - can use scdbg to emulate execution of shellcode
- shellcode2exe.py generates exe from raw binary format
    - -d supports shellcode encoded as %u Unicode string
    - -x supports shellcode encoded as \x hes string
- Jmp2it for when you can't emulate shellcode, runs directly w/out creating exe
    - jmp2it file.bin 0x0 pause   :allows you to pause and attach proc to x32dbg

- Shellcode conversion to binary format using Powershell ISE
    - F9 or right click / Toggle Breakpoint at shellcode line
    - Debug > Run/Continue
    - [io.file]::WriteAllBytes('file.bin',$var)   :cmd saves to raw binary file
    - scdbg can emulate running the bin file after

Manual code reversing

- Ghidra
    - *Ghidra data types available https://for610.com/ghidra-data
    - New Project, select file, then click on file for auto analysis
    - On analysis options make sure to enable option for "WindowsPE x86 Propagate External Parameters" (ids functions args named in MS API docs and list in comments)
    - Warning about file not containing debug info is normal
    - Useful to save Ghidra file and snapshot VM after importing
    - Note starting with Function Graph can help visualize function flow
    - Imports/Exports in bottom left Symbol Tree good place to also start. Note DLLs normal to have exported functions but exes normally don't have multiple. Import Address table helps direct code analysis. Available through Symbol tree (bot left) or Window / Symbol Refs
    - Remember Set Equate helps convert symbolic constants to actual value, do individually not for entire program
    - To clean up verb symbolic constraints can right click items like DAT, Data/ string ie.
    - Windows / Function Call Trees helpful to see calls made from current function
    - Functions tab (G key) can help id functions. Useful to rename interesting fnctns
    - Can be useful to view string references (Window / Defined strings) – i.e. may malware could reside in Application Data. String references shown w/Ctrl+Shift+F.
- X64dbg/x32dbg
    - Encrypted temp file example, Use PeStudio/Imports to spot API call to read file. X64dbg Command: *SetBPX ReadFile* sets a breakpoint at the read file start (case sensitive). Cmd: *run* runs until breakpoint. Look up function (ReadFile) parameters, hFile is 1st parameter & on x64 1st parameter is RCX – look at value (i.e. 110). Go to handles tab and find corresponding 0x110 handle. Could also look at malware properties using Process Hacker / Properties / Handles to confirm it reads the tmp file. X64dbg Call Stack tab shows where current function will return when finished, or Debug > Run to user code (or Alt+F9). In example saw CryptDecrypt. Clicked test eax,eax then Debug / Run until selection (or F4)
    - Areas with binary (i.e. MZ) can use Follow in Memory Map / Dump Memory to File
    - Areas with interesting functions, can set breakpoint on function and click EAX / Follow in Dump can keep an eye for interesting contents as stepped through
    - xAnalyzer can be used to display local vars/args (useful in unpacking)
- API Monitor
    - API Monitor lets you see what calls are made, whether a debugger type analysis like x64dbg or an active process. Better to launch malware from API Monitor if possible. First have to select the specific API calls to monitor, then the malware (process)
- OllyDumpEx
- Jmp2it
- Scylla

- IDA – popular disassembler w/built in debugger
- Windbg – popular & free Win debugger
- Radare2 – open source toolkit for Win & Linux
- Binary Ninja – commercial disassembler especially strong for automated analysis tasks
- Hopper - Commercial disassembler/decompiler that runs on OS X/Linux

Fully automated analysis

- Malware data repositories: VirusTotal, #totalhash
- Mulit-Engine AV Scanners: VirusTotal, MetaDefender, VirSCAN, AVCaesar
- File reputation: Malware Hash Registry, HashSets
- Automated Sandboxes: Any.run, CAPE, Intezer Analyze, Hybrid Analysis, more at https://for610.com/automated-sandboxes
- Website Investigation: vURL, Quttera, urlscan.io
- Other threat intel: PassiveTotal, Censys, Open Threat Exchange

Malware analysis Report Template: https://for610.com/report-mindmap

Initial Process: cut off internet. If internet req, use external and/or proxy/vpn (better than TOR). Spin up / destroy VPN servers every time & use different ip/regions. Check for DNS leakage.

Zeltser recommends first using OSINT, maybe already known bad / known good, then next static to help determine where to focus subsequent analysis efforts.

Import Library / API Keys Commonly Used by Malware

Self Defense

GetTickCount – could indicate that specimen has ability to measure speed of its execution which some malware uses to determine whether it's being debugged

Registry API Calls

RegSetValueExA – Capability to set registry values, common in malware
RegOpenKeyEx – Opens specified registry key, could be used in persistence
CryptDeriveKey – use of Windows cryptographic capabilities

File API Calls

ReadFile – If encrypted can step through to try to decrypt
GetTempFileName – creates a name for a temporary file, malware often uses this to name new files written to disk
GetTempPath – malware often uses temp paths on sys to write files

Encrypt/Decrypt calls

Crypt/Decrypt – Can be useful to step through debugger to decrypt encrypted files called by ReadFile

Possible C2 API Calls

InternetOpen/InternetConnect – Create HTTP Connection
HttpOpenRequest/HttpAddRequestHeaders – Build HTTP Request
HttpSendRequest – Send an HTTP request

InternetReadFile – Read a response

New Processes

CreateProcess – creates new procs; could spawn other procs

Possible Dropper Activity

*Rsrc section often used to store icons diag boxes, version info, but malware may hide exes here

FindResource – determines location of resource

SizeofResource – Obtain size of resource

LoadResource – loads .. resource

LockResource – Obtain pointer to resource

CreateFileA/WriteFile - CreateFileA returns handle to newly created file; WriteFile writes to disk

Can possibly extract w/PeStudio

Reinfection Check

CreatMutex – mutex often used to avoid reinfection

Monitor User Activities

GetKeyState/GetAsyncKeyState – Determines if particular key is pressed

GetWindowText – Retrieves text from a windows title bar

OpenClipboard/GetClipboardData/CloseClipboard-open for access, gather, then close

SetWindowsHookEx – Sets hooks, commonly used to monitor events like mouse movements, key presses, window interaction

Code Execution

ShellExecute – facilitates command execution

*VirtualAlloc often used by malware for unpacking

VirtualAlloc allows malware to create a new memory region on the fly

VirtualAllocEx can allocate memory inside another process

Code Injection

CreateToolhelp32Snapshot/EnumProcesses – Get list of running processes

Process32First/Process32Next – can iterate through procs

OpenProcess/CreateProcess – Opens a handle to targeted process/create new process

VirtualAllocEx – Allows a memory space in targeted Process

WriteProcessMemory – Writes specified contents (code/data) into mem or target process

*ReadProcMem+WriteProcMem together could be hooking indicator

CreateRemoteThread – Runs code in a new thread of the process (can involve LoadLibrary)

SetWindowsHookEx – Could be rootkit activity

Native APIs might be invoked as a way to try to be more stealthy:

CreateToolhelp32Snapshot/EnumProcesses – NtQuerySystemInformation

Process32First/Process32Next – NtOpenProces/ZwOpenProcess

OpenProcess/CreateProcess – NtAllocateVirtualMemroy/ZwAllocateVirtualMemory

VirtualAllocEx – NtAllocateVirtualMemory/ZwWriteVirtualMemory

WriteProcessMemory – NtWriteVirtualMemory/ZwWriteVirtualMemory

CreateRemoteThread – NtCreateThreadEx/ZwCreateThreadEx

Defenses

ScyllaHide plugin (x32dbg/x64dbg) – auto conceal debugger from common techniques

IsDebuggerPresent - returns non-0 if proc debuged, else 0

*best to replace code with NOPs. X64dbg / select JNE instruction/Assemble. NOP/Fill w/NOPs

Save the “patched” file after (File / Patch file / Patch File)

OutputDebugString – returns valid address only when being debugged

CheckRemoteDebuggerPresent – possible API call for anti-analysis defense

NtQueryInformationProcess – possible API call for anti-analysis defense
ZwQueryInformationProcess – possible API call for anti-analysis defense
BeingDebugged bit in PEB – instead of calling IsDebuggerPresent could check PEB bit
GetTickCount – malware can tell if exe too slow due to single stepping
GetLocalTime/GetSystemTime - malware can tell if exe too slow due to single stepping
NtAllocateVirtualMemory/ZwProtectVirtualMemory/ZwWriteVirtualMem-could be code inj
RtlDecompressBuffer – could be used for deobfuscation or unpacking

Process Hollowing/Replacement / RunPE
CreateProcess – or variants, launch suspended child process
Nt(/Zw)UnmapViewOfSection – Deallocates (hollows out) virtual memory of process
VirtuallAllocEx (/variants) – Allocates memory space in child proc for coming up code injection
WriteProcessMemory (/variants) – Writes new code to new allocated memory of child process
ResumeThread – Awakens child process to run injected code

Anti-Sandbox Techniques
Virtualization Check – Malware can check MAC address (OUI) or Video controllers; cores > 1; disk large
CountClipboardFormats – Dyre used to see if clipboard empty
GetCursorPos - Tinba used to see if mouse moving
GetForeGroundWindow – Tinba used; figured sandbox doesn't move active windows around
GetTickCount - Upatre checked to see how long has system been running
BlockInput – sometimes used to block debuggers & analysis tools
GetModuleHandleW (/variant) – can be used to look for security software (i.e. AVG)
FindWindow - can look for things like debuggers (ie OllyDbg)
tls-callbacks – PeStudio tls-callbacks / right click / disassemble; allows code run b4 entry point
LoadLibraryA / CC – if comparing first value of LoadLibraryA Int3 – checking for debugger

*Note malware often has similar chars; like it looks for memory address space (GetEIP & patterns), looks at loaded libraries. FS register has Thread Information Block, info about currently running thread. The pointer to Process Environment Block (PEB) which contains info about process including DLLs loaded or mapped to memory, in TIB at 0x30 offset; so the pointer is always at FS[0x30]

Registry Values Commonly Used by Malware

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\value
PendingFileRenameOperations reg key being blank can cause a file to be deleted
\Microsoft\CurrentVersion\Internet Settings - Internet Settings reg key can be malware modifying system internet connectivity settings, more frequently innocuous side off of pgoram instantiating a Windows library used for interacting w/websites
\Microsoft\Crypto\RSA – common side effect of process instantiating Windows library used to encrypt

Encode / Decode

Malware often uses XOR w/1 byte key to encrypt/obfuscate. Another common technique is rotate each byte in string by some number of bits to the right or left (ROR/ROL). Another is rotate alphabet characters (A-Z and a-z) by certain number of positions within alphabet .. seems worth writing a brute force and dictionary comparison. Incorporate all 3 then run freq.py and sort. Possibly even integrate w/tshark?

Loops in code could possibly be used for encoding/decoding, C2, port scan, DDoS, keylogger
xxd -r -p encoded.hex :convert hex to ASCII (sometimes in network traffic)
#Decode the URL example with the 5b key
xxd -r -p encoded.hex > encoded.raw; (convert hex to raw); translate.py encoded.raw decoded.txt 'byte ^ 0x5b'; cat decoded.txt
Alternative method: perl -pe ‘s/(..)chr(hex($1)^0x5b)/ge’ encoded.hex>decoded.txt
Unescaped(“%u4hex%u4hex%u4hex”) - common way save unicode to string; %u \x or other format
VBA common to see Hextostring / XORI for decoding
XORI can try to autoderive keys by examining plain text with encoded text
xorsearch -W -d 3 file.bin :can use XORsearch -W search common shellcode; -d 3 false pos
xorsearch -i -s file.exe http :-I case insensitive; -s generate file decodes all bytes using key
    *then when says Found XOR 41; strings file.exe.XOR.41 | more
brxor.py file.dll > file.txt : use brxor to deobfuscate XOR encoded strings that include
    english words without having to specify substrings
bbcrack.py -l 1 file.dll : bbcrack.py autoids patterns based on XOR ROL ADD
xorBruteForcer.py :look for encoded strings
NoMoreXOR.py :look for encoded strings
Xortool :look for encoded strings
unXOR :look for encoded strings
Kahu tools :look for encoded strings
oledump.py file.doc -p plugin_http_heuristics can decode common obfuscated url techniques
strdeob.pl 9.exe | more : looks for MOV insts operate on single byte val to build str
floss file.exe > file.txt :looks for clear-text/encoded strings

base64dump.py
    base64dump.py file.pdf :extracts/decodes base64 strings & more
    base64dump.py -s 2 -S : -s 2 specifies stream 2 and -S shows ASCII (-a is hex&ASCII)
    base64dump.py -m | more : supports several encodings besides base64
    base64dump.py file.txt -e pu :-e pu directs to look for strings using %Unicode %u
    base64dump.py file.txt -e bu : -e bu looks at backslash unicode

Register values

x32 Registers
EAX register used for addition, multiplication, return values
ECX register used as counter
EBP register used to ref args & local vars
ESP Register points to last item on a stack
EIP register points to next instruction to execute
ESI/EDI used by memory transfer instructions
EBX/EDX are generic registers used for various purposes
EFLAGS register represent outcome of computations and to control operation of CPU
Segment registers (CS,DS,ES,FS,FS,SS) describe different segments of memory
    Code Segment is default segment register when fetching instructions
    Data Segment is default register for accessing data w/ESI/EDI registers
    Stack segment default segment register for accessing data w/ESP register
*Note registers can be broken to 16 bit and 8 bit values (EBX, BX, BH, BL, ECX, CX, CH, CL, EDX, DX, DH, DL)

DLLs

Note DLLs need another EXE to load it into memory. Rundll32.exe can be used but may require user to specify exported function & supporting arguments.
Start by confirming file is DLL with PEStudio. Note ordinal column=1 – malware often exports/imports only by ordinal to make deciphering more challenging.
Ghidra / New non-shared project, load dll, initiate auto-analysis, save.
Symbol Table / exports. Entry is entry point for file, ex setting custom export that may point to functionality
In dropper example we can extract with PeStudio resources, dump raw

Web Downloads

Fiddler is a good option, remember to save TLS keys https://for610.com/tls-under-hood
Run Process Monitor before downloading, watch for sub/spawned processes.
Run Wireshark in addition to Fiddler before starting, look for traffic not seen in Fiddler (i.e. DNS req)
Malicious scripts often obfuscated to evade detection and complicate analysis – deobfuscate by allowing them to decode themselves under controlled conditions
Notepad++ plugins like JSTool (JSMin/JSFormat) help with beautification making easier to read
If debugging in IE add debugger; in front of <script> to activate. Inspect variables, see call stack, specify variables to watch. Console also available. Useful to set homepage to about:blank so debugger doesn't complain (F12 brings up Developer Tools / Debugger tab). Insert breakpoint in front of what you want to see (i.e. variable). Advantage of IE is you can set a breakpoint in the middle of a line. Once you step through to break point go to console, type console.group(var) to see the variable var. Could also see in Watches region. Escaped quotes \" need to be replaced with regular quotes. Console might add stuff like undefined – also need to remove when copying over to Notepad++.
Script interpreters (SpiderMonkey/CScript/V8) can execute script outside browser. Sometimes easier than debugger because we don't need to set break points, but do need to define objects & properties referenced by script. Note SM/V8 good with IE, CScript good w/other browsers.

JavaScript

box-js helps w/javascript; --download allows script to download (default is to not)

```
box-js file.js                                       :default run
box-js file.js --no-shell-error --no-file-exists>out.txt:default tries to branch on error
```

Possible signs of obfuscation (Good for debugger break points):

- document.body.appendChild
- document.parentNode.insertBefore
- document.write
- eval

PDF Counter Techniques

app.setTimeOut tricks some analysis tools, indirect way of launching function, can delay execution of attack until doc fully loaded to PDF
app.viewerVersion ids version of PDF viewer being used; could be used for anti-analysis
syncAnnotScan/getAnnots retrieves annotations embedded in doc, enable script to store some contents as annotations and to assemble script during runtime. Similar to getElementById in browsers

arguments.calle is antidefense, checks if modified. Avoid modifying functions that call arguments.callee

DeObfuscation techniques:

var = (8f2, 233, 44, 5f6, 36, “it”);	:js tuples assign *last* value to var
var=(02.e1>=4e1?.9075:”i”+”f”);	:if then else basically assigns if to var
var b=window; var xy=b[esv8asls’.replace(/[sDHt8]/g,’’)];xy(a);	:regex window.eval
<input type='hidden' id='key'…	:hidden in html; document.getElementByID retrieves
rev file.js > file.txt	:reverses strings for files with reversed cmds

js-beautify helps clean up JavaScript for easier reading

PDFs

Risky keywords

Execute embedded JavaScript: /JS /JavaScript /XFA
Run an embedded Flash program: /RichMedia
Launch external or embedded program: /Launch /EmbeddedFiles
Interact with websites: /URI /SubmitForm
-EncodedCommand: Base64 string

pdfid/pdf-parser

pdfid.py file.pdf summarizes risky aspects of PDF file to begin assessment; --extra

pdf-parser.py file.pdf --search JavaScript	:search using prev id’d keywords

*note --search doesn't look inside streams; need to use --searchstream parameter

pdf-parser.py --object 10	:search objects ref’d by JS
pdf-praser.py file.pdf --object 13 --filter --raw -d output.txt	:filter decodes stream; raw no escape chars

peepdf (alt to pdfid/pdf-parser)

peepdf.py -fli file.pdf	: -i interactive; -f force parsing mode, -l loose parsing mode
info	:info shows suspicious elements
object 12	: object <12> cmd looks at specified object
stream 13 > file.txt	:saves stream to file.txt

base64dump

base64dump.py file.pdf	:extracts/decodes base64 strings & more
base64dump.py -s 2 -S	: -s 2 specifies stream 2 and -S shows ASCII (-a is hex&ASCII)
base64dump.py file.js -s 10 > script.ps1	:dump the 10th id (for example) to base64 decoded form
base64dump.py -m | more	: supports several encodings besides base64
base64dump.py file.txt -e pu	:-e pu directs to look for strings using %Unicode %u
base64dump.py file.txt -e bu	: -e bu looks at backslash unicode

Stream can encode contents; decoded by applying 1 or more filters

*Password protected – structure visible but streams need decryption to see

Office Macros

oletools

olevba.py file.docm | more	:extract macros from docs; even if passwd protected VBA not

OOXML only allows extensions ending in m (.docm) to execute macros; OLE2 format not so much

```
Unzip file.docm -d media                 :unzip Office docs to media folder
file file.bin                            :*Composite Document File V2 is a OLE2 file
strings file.bin | grep http             :look for interesting strings
strings --encoding=l file.bin |grep http :also look for Unicode encoded strings
oledump.py file.bin/file.docm            :M indicates VBA macros
oledump.py -s 3 -v file.bin/docm|more    :examine macro stream 3 id'd in earlier step (-v decompresses)
```

SRP Streams – contain cached versions of earlier macros
VBA common to see Hextostring / XORI for decoding;
XORI can try to autoderive keys by examining plain text with encoded text
StrReverse can be used for obfuscation : rev file.js > file.txt
oledump.py file.doc -p plugin_http_heuristics can decode common obfuscated url techniques

Password protected:

```
oledump.py file.doc                      :identify macro stream
oledump.py file.doc -s 7 -v > macro.txt  :assuming stream 7, output macro to macro.txt
```

* alt tools olebrowse/Ssview let view/extract OLE2
*When running obfuscated macros in Office set breakpoints (i.e. AutoOpen, CreateObject, etc), look at Locals window for useful variable contents

p-code
*pcodedmp might not handle all versions of Office; p-code intricacies are specific to each Office version
Running p-code to look at behavioral analysis requires the corresponding version of Office

message_extract.py

```
message_extract.py file docm extracted.exe       :extract exes from macros

pcodedmp.py -d file.doc                  :oledump can't handle p-code well
pcodedmp.py file.bin > out.txt           :p-code can be contained in bin files
```

RTFs

RTF's don't support macros, but still subject to OLE1/other binary contents
Object data is searlized to a bytes string; embedded object \objdata

```
rtfdump.py file.doc                      :look at various levels, look at length value (l)
rtfdump.py file.doc -s 7  | more         :say level 7 has large, might want to look at it; also look for
groups w/deep nesting levels w/many hex chars h= (embedded objs/binary stored as serialized strings);
also b= tag search how many \bin entries (can be used to conceal risky contents)
rtfdump.py file.doc > out.txt; wc -l out.txt :look at # of groups; many indicate obfuscation
rtfdump.py file.doc -f O                 :-f O groups that enclose objcts; look for high hex (h=) cnt
rtfdump.py file.doc -s 5 -H -d > file.bin :-H converts from hex form; -d dumps to raw binary format
rtfdump.py file.doc -s 166 -H -c 0xBA0:0xCE0 -d > out.bin          :carve out files
rtfobj.py file.doc; frtfobj.py file.doc -s 0  :alternative carving solution
xorsearch -W -d 3 file.bin               :can use XORsearch -W search common shellcode; -d 3 false pos
xorsearch -i -s file.exe http            :-i case insensitive; -s generate file decodes all bytes using key
        *then when says Found XOR 41; strings file.exe.XOR.41 | more
```

brxor.py file.dll > file.txt : use brxor to deobfuscate XOR encoded strings that include english words without having to specify substrings
scdbg start at offset from xorsearch :for ex if ids GetEIP; that's malware seeing where at in mem

Zip Files

Zipdump examines zips including XML based Office docs

zipdump.py -y file.yara -C decoder_xor1,decoder_rol1,decoder_add1 file.xlsx :similar to xorSearch -W
zipdump.py -e file.xlsx : look for null control, high bytes (last column)

Packers

Indicators of packers: few readable strings; high entropy, few imports, packer strings
*VirtualAlloc often used by malware for unpacking
Remember to *disable ASLR* before unpacking (ASLR randomizes image-base) – find DllCharacteristics field in PE header & disable DynamicBase flag in it.
    CFF Explorer – Optional Header on left section / DllCharacteristics row, click here. Uncheck DLL can move field, ok. Save.
    Setdllcharacteristics -d file.exe :alt to CFF Explorer
One way to tell if unpacked yet (when stepping through) is look at string references

unpack / id packer
upx -d :can bypass protection; but variants of UPX scramble
Ether/UnpacMe/TitanMist :other unpackers to autohandle
PeStudio :look@imports, sctns names (ie UPX/uncmmn)/entropy high/rawsize0 vsize>0
Bytehist can help show byte usage histograms to id packed (more uniform) vs not (less uniform)
Detect It Easy / Exeinfo PE :try to identify packers used (diec file.exe)
Alt packer id: trid, file, pepack, packerid, pescan, ProtectionID, RDG Packer Detect, CFF Explorer

Strings in memory
Strings in memory is a good place to start: Process Hacker / Properties/Memory/Strings

Unpack process from memory (only getting IAT, but not OEP)
*Remember to have disabled DynamicBase 1st to disable ASLR (setdllcharacteristics -d file.exe)
Scylla, PE Tools, Universal Import Fixer, Imports Fixer. Rebuild IAT, mindful of Original Entry Point
Scylla, exe, Dump. (cleared dynamicBase flag prior, to disable ASLR prior).
Then IAT Autosearch, Get Imports, Fix Dump to populate IAT. Generally good for IAT but not entry point. To find entry point need to do in a more nuanced manner. Note that rebuilding IAT lets you see API calls and perform static analysis to look at decoded code and strings, but without entry point you won't be able to execute code and do behavioral analysis.

Slower approach to unpack process from memory (to get both IAT & OEP)
*Remember to have disabled DynamicBase 1st to disable ASLR (setdllcharacteristics -d file.exe)
Load packed specimen in debugger like x64dbg
    -x64dbg will pause at Entry Point
Locate the end of the unpacker & set breakpoint
    -This can be time consuming and challenging w/some packers
    -For UPX, scroll in disassembler until JMP near end of code prior to area filled with zero opcodes

-Set breakpoint (F2) at end of unpacker

Run specimen to let it unpack original program to memory and pause at end

-F9 is run or just Debug/Run. Takes you to breakpoint.

Single step to let process jump to unpacked code (OEP)

-Then step into/over(F8) – F8 jump should take you to OEP

-Confirm by right click, Search Current region. Look for String refs & Intermodular calls (APIs)

Dump unpacked process

-x64dbg/plugins/OllyDumpEx/Dump process – dumps process from memory

Fix up PE header, paying attention to IAT & EP

-OllyDumpEx button Get EIP as OEP (OllyDumpEx calcs OEP = imagebase-EIP(RIP on x64))

-Sometimes some packers you have to enable MEM_WRITE for sections (OllyDumpEx window, double click row for Section where mem violation crashed when you tried to run output

-Dump / Finish

Fix IAT table

-Scylla plugin for x64dbg; IAT Autosearch, No, Get Imports, Fix Dump & point orig to output

Examine in Memory without Dumping

Could set breakpoints on API calls we know unpacked call will make (i.e. SetBPX, ReadFile) but meh
*Remember to have disabled DynamicBase 1st to disable ASLR (setdllcharacteristics -d file.exe)

clears break points (in x64dbg)

cleardb (cmd)

look for memory segments w/E flag in Protection column. Probably in one

run (F9)/memory map tab

Next eliminate nonexecutable segments and standard windows DLLs to narrow down (still E flag) –

Right click / Follow in Disassembler

One strategy is to look at API calls:

Right click, Search for/Current Region/Intermodular calls (opens subtab Calls)

In calls tab find interesting APIs like crypt/decrypt, right click / Follow in Disassembler

Set *hardware* breakpoint (F2) after the call if you're interested in looking at results of code

-software breakpoints likely to be forgotten when we restart process

Restart process (Cntrl+F2), else if we just tried running it wouldn't work because already running

Memory Forensics

Software for dumping (can displace useful mem contents): WinPMEM, Comae (DumpIt), BelkaSoft RAM
Analysis tools (free): Volatility, Rekall, Redline

Volatility

```
vol.py --help                                          :show various commands
vol.py -f file.vmem kdbgscan | more                    :determine likely profile; "Profile suggestion"
vol.py file.vmem --profile=win10x86_10586 pslist       :run pslist to see if right profile picked
vol.py file.vmem --profile=win10x86_10586 pstree       :pstree shows parent-child relationships
        *easier to spot anomalies if you know how sys looked before infection
export VOLATILITY_PROFILE=Win10x86                     :set environment variable
vol.py -f file.vmem cmdline | more                     :see command line details
vol.py -f file.vmem memdump -p 872 -D /tmp                 :save vritual mem of designated proc
vol.py -f file.vmem ldrmodules                             : you could use dlllist to look for DLLs injected
vol.py -f file.vmem dlllist                                :look for DLLs injected if DLL unlinked
vol.py -f file.vmem malfind -D /tmp> malfind.txt           :xmin  virtual address descriptor; often flse pos
vol.py -f file.vmem apihooks       :looks for user/kernel hooks --quick=crit DLLs; --skip-kernel is user only
```