SANS FOR610 | Reverse-Engineering Malware: Malware Analysis Tools and Techniques

This is my notebook for the SANS FOR610 course in preparation for the GIAC GREM exam. The SANS FOR610 course covers both core reverse-engineering topics and in-depth malware analysis techniques. This notebook should be useful for anyone else looking to break into the world of reverse-engineering malware.

The reverse engineering toolkit used for the course can be found on my Google Drive here:

- https://drive.google.com/open?id=1MiH0sAbPP_gZPUa7ej6HUhiKQ-9dWId1

The malware specimens are compressed and encrypted. The password to decrypt and decompress specimens is:

```
malware
```

You can also download and find the documentation for the **REMnux** reverse-engineering toolkit at:

- https://remnux.org

Please feel free to let me know if there are any errors in the Markdown notes provided, or if information contained within is invalid or incorrect.

FOR610.1: Malware Analysis Fundamentals

Introduction to Malware Analysis

Modern malware is used to remotely control the compromised system, spread within the organization, exfiltrate sensitive documents, spy on the victim, etc.

Stages of Malware Analysis

- *Fully automated analysis* - used to quickly assess what the specimen might do if it ran on a system; produces reports showing mutexes, registry keys, network traffic, etc.
- *Static properties analysis* - analysts review the metadata of a malware specimen; reviewing strings of an embedded file, overall structure of the specimen, header data; without running the actual program
- *Interactive behavior analysis*
- *Manual code reversing*

Malware analysts need to communicate with other actors within the organization effectively to triage, enumerate, catalog, and protect the organization from malware. Malware analysts need to receive inputs from other security professionals such as:

- verbal reports
- suspicious files
- file system images
- memory images
- network logs
- anomaly observations

and malware analysts must output to the community and the organization formatted reports for the community to digest. These reports can encompass:

- what a specific malware specimen does
- how to identify a malware specimen
- attacker's profile
- IR recommendations
- reports and indicators of compromise (IOC)s
- malware trends

What to include in a malware analysis report

- *Summary of the analysis* - executive summary; key takeaways; malware specimen nature, origin, capabilities, and other relevant capabilities
- *Identification* - type of file, name, size, sha256sum, and current antivirus detection capabilities

- *Characteristics* - capabilities for infecting files, self-preservation, spreading, leaking data, interacting with the attacker, etc.
- *Dependencies* - resources required for the malware specimen to operate; supported OS versions, .dlls, .exe, URLs, and scripts
- *Behavioral and code analysis findings* - overview of the specimen behavior; static and dynamic analysis observations
- *Supporting figures* - logs, screenshots, string excerpts, functions listings, and other exhibits that support the analysis
- *Incident recommendations* - indicators for detecting the specimen on other systems and networks, and possible eradication steps

Running strings and googling hashes, resources, etc. identified being used by a malware specimen is called *Open-source intelligence (OSINT)*. Using tools to visit suspicious URLs, searching for IOCs in databases, and following the location of where the malware specimen beacons to are parts of OSINT - all available on the internet.

Network traffic used by malware

- *Beaconing* - sending brief periodic messages to the adversary with basic information about the state of the malicious program and its infected host
- *Command and control* - obtaining instructions from the attacker via the network
- *Exfiltration* - sending stolen files or data (such as keystrokes) over the network back to the attacker

When utilizing OSINT, be hesitant to upload files to a third party that seem suspicious and might not be captured in a database, yet. This might tip off the attacker that they have been discovered. Sending hashes for review is acceptable, but unless the virus seems well known, use discretion.

Malware Analysis Lab

Despite all of the open-source tools available, sometimes your organization will require you to keep a breach under wraps. Also, the malware you have encountered might not have been discovered, yet. There might not be any OSINT available related to your particular infection.

In order to closely study the malware on your own, execute it and record its behavior, controls its resources, and prevent it from infecting other victims, you must construct a lab environment.

The malware analysis should comprise multiple systems networked together, and you should have a mix of Windows and Linux operating systems. In this class, we will be analyzing malware that targets Windows, however, Linux can provide the network services that the malware might be expecting.

You should pay attention to your lab isolation measures. There is still risk associated with running the malware within a virtual machine. Especially sophisticated malware specimens can attempt to escape virtual machines or utilize resources provided (such as the network or shared file systems)

to infect other portions of the lab environment or the host machine, itself.

Precautions one should take for lab isolation:

- Keep up with security patches for your virtualization software
- Don't use lab systems for other purposes
- Disconnect the lab from other networks
- Disable risky capabilities, such as folder sharing

Malware might also try to determine if its being analyzed. It will attempt to detect virtualization, debuggers, monitoring and analysis tools and attempt to obfuscate its code or fool analysts. It is also possible for it to interfere with analysis tools, terminate its execution, or just exhibit different characteristics entirely.

At the end of the day, you might need a physical system to run a particular piece of malware. Ensure to use tools such as `dd`, `ddp`, `clonezilla`, or `pxe` for reverting back to the last-known-good physical state of a machine.

The lab should include tools that can examine the specimen statically and dynamically from several vantage points.

Here is a listing of some important tools for static analysis:

- PeStudio
- strings
- CFF Explorer
- peframe
- Detect It Easy
- HxD

Here is a listing of some important tools for behavioral analysis:

- Process Hacker
- Process Monitor
- RegShot
- Wireshark
- fakedns
- TcpLogView

Here is a listing of some important tools for code analysis:

- IDA
- x64dbg / x32dbg
- OllyDumpEx
- jmp2it
- Scylla

Static Properties Analysis

Before conducting behavioral analysis of a malware specimen, it's best to start with reviewing the static properties of the suspicious file.

The things we should be looking for are:

- file and section hashes
- packer identification
- embedded resources
- imports and exports
- crypto references
- digital certificates
- "interesting strings"

This will help us answer these questions:

- Is it malware?
- How bad is it?
- How to detect it?
- How to analyze it?

Multiple different static analysis tools exist to extract ASCII and Unicode strings from a file. This allows the analysts the ability to make inferences on the nature of a particular specimen. Usually you can draw the registry keys, mutants, User-Agents, and network locations being called out to via strings.

There exist multiple different software packages for static analysis of specimens as well. These packages can outline the .data, .rsrc, .text, and .reloc portions of a portable executable and provide an determination about their possible maliciousness. You should be on the lookout for specific API calls that could be used to indicate malicious behavior.

There are also tools to detect if a specimen has been packed. These tools can identify the packer used to create a specific piece of malware and extract the original program from the data portion of the packed malware.

Behavioral Analysis Essentials

Several tools for Windows exist in order to allow analysts to capture how a piece of malware behavior when executed. Here a list of some of the tools used in this section of the class:

- Process Hacker
- Process Monitor
- Regshot
- ProcDOT

- Wireshark

In the exercise for this section, we run `brbbot.exe` as Administrator in order to observe its behavior. We run `Process Monitor` and `Process Hacker` to record the specimen's actions, `Regshot` to take a snapshot of the registry prior to running `brbbot.exe`, and `Wireshark` on the upstream Linux host in order to record the network traffic `brbbot.exe` generates.

NOTE: It's best not to run Wireshark on the host that you plan to run a malware specimen on. The malware specimen could use this to detect if it's being observed.

In this exercise `brbbot.exe` was unable to resolve a hostname, presumably its callback location, and so thus it exited execution. In order to trick specimens into thinking they have internet connectivity, we can use a tool like `fakedns`. This tool will response to a specimen's DNS requests and provided it fake name resolution.

Code Analysis Essentials

IDA/IDAPro, x64/32dbg. Using x64dbg, set breakpoints for interesting API calls using the command-line command `SetBPX`. You can view `handles` of a malware specimen using the `handles` tab of x64dbg, or you can view the `handles` using `Process Hacker`.

For 64-bit architectures, according to Window's documentation, the pointers and integers being passed to a Windows API call will be located in the `rcx`, `rdx`, `r8`, and `r9` registers - in that order.

Another effective way of watching a process execute suspicious API calls is by using `API Monitor`. `API Monitor` is a free tool in which you can specify which Windows API calls to trigger on for a specific process. You can attach `API Monitor` to a running process, or execute the specimen with `API Monitor`.

Interactive Behavioral Analysis

In this exercise we have an encoded hex file that seems to be XOR'd with the value `5b`. In order to return this file back into binary so that we can XOR each bit, we use the `-r` option for `xxd` to do the reverse operation that `xxd` provides.

We have a tool on the REMnux distro called `translate.py` that is designed to conduct specified bitwise operations on a file (XOR, ROR/ROL, etc).

In this exercise, there's a specimen, `juice.exe` that attempts to reach out to multiple hard-code IP addresses, avoiding the use of hostname-resolution. This technique renders our use of `fakedns` inert, thus we use `iptables` to create a `PREROUTING` rule that forwards all traffic to our local ports. This is a great technique for making sure you capture all network traffic and forward it to a network device for analysis.

Additional tools are available for intercepting and analyzing network connections:

- TcpLogView
- PE Capture
- ApateDNS
- FakeNet-NG

Class Notes

Tools to look at:

- **pestr** - better than strings as strings only looks at ASCII text, however, *pestr* also views Unicode strings and can filter for malicious patterns.
- **BinText** - GUI tool to view embedded strings on a Windows OS.
- **strings2** - Windows command-line tool for viewing both ASCII and Unicode strings on both static applications, as well as currently running processes.
- **PeStudio** - static analysis of compiled C++ and flag anomalies in the binary.
- **gchq.github.io/CyberChef**

Tools to look at:

- CAPE Malware Analysis and Payload Extraction. (https://cape.contextis.com/analysis)
- urlscan.io (allows you to scan a suspicious url found in malware)
- any.run (cloud procured VMs for dynamic malware analysis)
- rapid_env (https://github.com/adamkramer/rapid_env)
    - Rapid deployment of Windows environment (files, registry keys, mutex etc) to facilitate malware analysis
- threatconnect (freeware malware callback analysis)
- intezer (genetic malware analysis)
- packetflow
- detectionlab (clong/detectionlab github)

Definitions

- **Malware** - code that is used to perform malicious actions; designed to allow the attacker to benefit at the victim's expense; malicious purposes
- **Open-source intelligence (OSINT)** - information freely available about a specimen on the internet; gathering information about a specimen using tools available online
- **Indicators of Compromise (IOC)s** - specific signatures for a malware specimen that indicates its existence on or infection of a system.

610.1: Malware Analysis Fundamentals

affiliate id (affid)	used to identify an infection campaign or the entity that is distributing the malicious program
apateDNS	dns server for redirecting hostname resolution requests; similar to fakedns but runs on windows
beaconing	sending brief, periodic messages to the adversary with basic information about the state of the malicious program and its infected host
behavioral and code analysis findings	part of a formal malware analysis report; overview of the analyst's behavioral, static, and dynamic analysis observations
Binary Ninja	a commercial disassembler that's especially strong for automated analysis tasks
BinText	provides an interactive and flexible GUI for examining embedded strings on Windows
botnet	malware that "calls home" to a command and control center for further instructions after it infects a computer
characteristics	part of a formal malware analysis report; specimen's capabilities for infecting files, self-preservation, spreading, leaking data, interacting with the attacker, etc.
clonezilla	disk cloning software enabling the analyst to save the laboratory system's hard disk image and then reapply it after completing the analysis
command and control	obtaining instructions from the adversary regarding actions that the specimen needs to perform
CryptDeriveKey	indicates that the specimen leverages Windows cryptographic capabilities
ddp (delta-delta-patch)	used to create a patch from an existing dd image and then re-apply it
dependencies	part of a formal malware analysis report; files and network resources related to the specimen's functionality - supported OS versions, required initialization files, custom DLLs, executables, URLs, and scripts
detect it easy	a useful Windows tool for determining what tools were used to generate the specimen; examines the PE header
disassembling	involves translating binary machine-level instructions to human-readable assembly code
dynamic code analysis	involves examining the code at the assembly level while running the program
exeinfo PE	another useful Windows tool for determining what tools were used to generate a PE specimen; examines the header of the file
exfiltration	sending stolen data, such as keystroke logs, to the adversary
exiftool	displays metadata embedded in various file types
fakedns	responds to all hostname resolution queries and provides the IP address of the reverse engineering virtual machine; can be used to trick malware into thinking it has network resources
fakenet-ng	intercepts network traffic in the lab, emulates common protocols, similar to inetsim but runs on Windows
fiddler	a tool that can intercept and automatically generate responses to HTTP and HTTPS requests - client-side
fully automated analysis	used to quickly assess what the malware specimen might do if it ran on a system; produces reports showing mutexes, registry keys, network traffic, etc.
handle	in Windows, a handle is similar to a file descriptor; a handle points to the actual resource being used by a process
hopper	a commercial disassembler and decompiler that runs on OS X and Linux
IDA	renowned disassembler for static code analysis of binary executables
identification	part of a formal malware analysis report; type of file, name, size, hashes, and antivirus detection capabilities
imports	Windows uses this section of an executable to determine which DLLs and the functions implemented within them (symbols or APIs) are necessary for a program's execution.
incident recommendations	part of a formal malware analysis report; indicators for detecting the specimen on other systems and networks and possible steps for eradication
indicators of compromise (IOCs)	an artifact observed on a network or in operating system that with high confidence indicates a computer intrusion; represents intrusion signature; IDS can be tuned to watch for the signature to prevent future compromise

31. **inetsim**	a tool used to emulate the common protocols HTTPS, SMTP, FTP, POP3, TFTP, and IRC; can be used to fool malware trying to use more sophisticated measures of reaching the internet
32. **interactive behavior analysis**	running the malware in a test environment; providing the malware with resources at each stage of its execution to see how it behaves
33. **iptables**	powerful Linux-based firewall software; we can use it to intercept and redirect network connections
34. **LoadLibraryW**	indicates that a specimen can load additional DLLs during runtime
35. **malware**	code that is used to perform malicious actions, typically designed to allow the attacker to benefit at the victim's expense
36. **manual code reversing**	disassembly of a malware specimen to determine, at the lowest level, how it is intended to operate and how it behaves
37. **MASTIFF**	extracts many details from various types of malware; good for bulk review of many samples
38. **mutant**	sometimes referred to as a mutex, this serves as a flag that programs can use to serialize access to a resource; sometimes used by malware to avoid reinfecting the host
39. **open-source intelligence (OSINT)**	gathering information from public data sources
40. **packing**	typically involves obfuscating, encrypting, or encoding the original executable file to create a new file that embeds the original program as data; when the program runs the original program is unpacked
41. **patching**	editing compiled executables to prevent the specimen from conducting a specific branch of code execution
42. **PE Capture**	records and captures local PE files that try to run
43. **peframe**	an open source tool to perform static analysis on Portable Executable malware and generic suspicious files
44. **pescan and portex**	examine key aspects of Windows executable files and identify anomalies
45. **pescanner.py**	a PE analyzer written in python by the authors of the Malware Analysts Cookbook
46. **pestr**	Strings analysis tool on REMnux; designed for extracting strings from Windows executable files - obtains both ASCII and Unicode-encoded strings
47. **PeStudio**	Provides an analysis of the static properties of a portable executable; Windows tool; calculates various hash values for indexing a specimen; outlines indicators of malicious activity for a specimen
48. **pivoting**	looking for associations between known attributes of the malicious program with new characteristics
49. **ProcDOT**	visualizes Process Monitor logs for easier analysis
50. **Process Hacker**	open-source tool; GUI designed to help analysts monitor system resources, debug software, and detect malware - replacement for Task Manager
51. **Process Monitor**	Sysinternal tool, shows real-time file system, registry, and process/thread activity - records all observed actions in a log file
52. **PXE (preboot execution environment)**	Refers to a client that can boot from a NIC. PXE-enabled clients include a NIC and BIOS that can be configured to boot from the NIC instead of a hard drive. It is often used to allow clients to download images.
53. **r8**	this is the third register passed to a Windows API call
54. **r9**	this is the fourth register passed to a Windows API call
55. **radare2**	open-source toolkit for Windows and Linux, installed on REMnux
56. **RCX**	this is the first register passed to a Windows API call
57. **rdx**	this is the second register passed to a Windows API call
58. **RegSetValueExA**	indicates that a specimen has the capability to set registry values
59. **Regshot**	highlights changes to the file system and the registry
60. **signsrch**	locates code used for crypto, compression, and more
61. **snapshot**	saving the state of the virtual machine in order to revert back to a last-known-good if the malware destroys the lab environment

62. **static code analysis**	involves using a disassembler to examine the program's code without actually executing it
63. **static properties analysis**	examining a malware specimen by reviewing its metadata; looking at strings, structure, and header data without actually running the program
64. **strings**	Tool present on most Linux distributions - by default only extracts ASCII-encoded strings; use --endcoding=-l to extract Unicode strings; use the -a parameter to scan the whole file
65. **strings2**	Command-line tool for extracting strings on a Windows system; extracts both ASCII and Unicode strings; can extract strings from a running process
66. **summary of the analysis**	part of a formal malware analysis report in which the writer provides the key takeaways to the reader; specimen's nature, origin, capabilities, and other relevant characteristics
67. **supporting figures**	part of a formal malware analysis report; logs, screenshots, string excerpts, function listings, and other exhibits to support the report
68. **tcplogview**	maintains a historical log of local TCP connections, showing which process handled which connection
69. **trid**	identifies the type of file you're trying to examine
70. **viper**	manages the malware collection and extracts various static properties about the files
71. **windbg**	powerful and free Windows debugger from Microsoft
72. **Wireshark**	Application that captures and analyzes network packets
73. **x64dbg / x32dbg**	open-source debugger for Windows
74. **xxd**	tool used to dump binary files into readable hex

FOR610.2: Reversing Malicious Code

Core Reversing Concepts

While behavioral analysis is useful in initially determining the capabilities of a malware specimen, code analysis will allow the analyst the ability to accurately examine all branches of execution and provides a comprehensive view of all malicious functionality.

The primary disassembler we use for this course is **IDA**. IDA is a recursive traversal, interactive disassembler - more accurate and thorough than disassemblers that conduct linear sweeps.

IDA uses a technology called **FLIRT (Fast Library Identification and Recognition Technology)** to automatically identify common libraries used within an executable under analysis.

The **Exports** tab in IDA displays the entry point of executables or the locations of multiple exported functions. The **Imports Address Table (IAT)** in IDA displays the APIs used by the program that are contained in external libraries. Viewing the API calls that a malware specimen uses can allow the analyst to infer the specimen's capabilities, functionality, and intent. Windows malware often interacts with the registry to configure itself for persistence or store configuration data - we should always investigate changes to the registry.

To find all of the instances of an API call in the disassembled code, double-click the API call in the Imports Tab to travel to its location. Then right click the API call to find the option:

> Jump to xref to operand...

Or you can press x on the keyboard to determine all cross-references of the API call in the disassembled code.

In order to effectively reverse-engineer disassembled code, we need to understand how to read assembly code effectively. You can find my definitions / flash cards for the different registers of the Intel x86 architecture in the **quizlet** portion of this repo.

Direct memory addressing is pretty straight-forward. Also works with pointers to memory. IDA usually shows the de-referencing of a pointer by annotating it as such:

> _

Indirect memory addressing is a little more complicated. We calculate our effective memory address by using some base register, an index and a scale, and then the displacement. Some examples:

- `[eax]` - access dynamically allocated memory using just the base register
- `[ebp + 0x10]` - access data residing on the stack (base + displacement)
- `[eax + ebx * 8]` - access an array with 8-byte structures (base + index * scale)

- `[eax + ebx + 0xC]` - access fields of a two-dimensional array of structures (base + index + displacement)

When reverse-engineering a malware specimen, we must keep cognizant of the "code-data duality" that exists in computing. When looking at information, we must take into context what the data represents as code can be represented as raw data, and vice versa. This is how malware can obfuscate and unpack itself so well - we won't know what the data represents until runtime.

IDA has the ability to change the current representation of values in the disassembled code. IDA will display the hex values of constants being passed to API calls, however, after right-clicking a value of interest, the analysts can request IDA to represent the value as a standard symbolic constant. This will allow the analyst to choose from a list of matching symbolic constants, but these constants are usually from a list of macros most likely defined in the header file included to compile the target binary.

Subroutines within IDA are represented as such:

> sub_location

You can view all of the function calls made by a particular subroutines by using this IDA feature:

> Go to View > Open subviews > Function calls

Doing this can help an analysts navigate large subroutines as well as infer the purpose of the current subroutine being reviewed.

Reversing Functions

We first begin with the function prologue and epilogue. This will help us understand what takes place before and after a function is called, and how the subroutine has access to variables / data required to complete its operations.

The function prologue occurs at the start of a function. Here, the function will allocate space for variables, and save registers that will be reused in the function body. Function arguments get pushed to the stack, and the stack pointer is saved for reference when the function is returned to its caller.

The function epilogue occurs after the function is complete. The epilogue cleans up the stack and restores the registers and the information they contained prior to calling the function.

The following are some good questions to ask yourself about any function you hope to reverse engineer:

- From how many locations is it called?
- How many arguments does it take?
- How many local variables does it use?
- Which instructions comprise the prologue?

- Which instructions comprise the epilogue?
- What calling conventions does it use?

Final notes for this section:

- The stack is used to store arguments and variables.
- Understanding stack details explains how code can follow numerous branches but always return successfully.
- Calling conventions dictate how parameters are passed and how stack cleanup occurs.
- Identifying the function prologue and epilogue is key to separating "administrative" code from core functionality.

Control Flow In-Depth

If-Else statements in assembly, translated from C/C++, usually have an initial code block at the `if` statement that must conduct a comparison between two values and issue a conditional jump instruction. If an `if` statement fails to evaluate to `true`, the code will then issue an unconditional jump to the second condition statement, `else if`. Finally, the an unconditional jump will be issued to the `else` statement if all other statements fail to evaluate to `true`.

So we've been using the Imports table to infer the nature and capabilities of a malware specimen. We can also determine the nature of a malware specimen by reviewing its strings. By default, IDA shows the ASCII strings of a decompiled binary, however, we can more thoroughly review the strings by including the UTF encoded strings as well. Do the following in IDA in order to view UTF encoded strings:

> View > Open subviews > Strings

Then modify the IDA configuration to include Unicode strings:

> Right-click the IDA Strings window > click "Setup" > check the box for "Unicode C-style (16 bits)"

Loops usually appear in malware for these reasons:

- Encrypt / decrypt network traffic - loop over each character in the string to send
- Attempt to connect to C2 servers - loop over a list of servers
- Perform a port scan - try to connect to port 1 - 65535
- Perform a DDoS attack - keep sending malicious packets
- Log keystrokes - check state for each key code 0 .. 92.

Looping methods:

- Utilize a conditional jump to repeat execution.
- Utilize `loopxx` instructions:
    - Examines the `ecx` register

    - Automatically decrements the `ecx` register
    - `loopz`, `loopnz`, `loope`, `loopne`

`loopxx` have a maximum jump range of 128 bytes.

You can determine what type of condition statement you are reviewing in assembly by checking when conditional jumps are executed. If a conditional jump jumps past another comparison due to a value being interpreted as `false`, it's a safe bet that you're looking at an `and` statement. The opposite is true for an `or` statement - the first time something is `true`, you'll probably jump to the rest of the code block.

Complex condition statements will most likely have multiple comparisons, conditional jumps, and code blocks to be executed. It's recommended you keep a worksheet handy for reversing condition statements so that you can sketch the logic into a flowchart.

Lastly, switch statements can be identified in assembly by the use of jump tables by the compiler. A variable is usually evaluated for a specific range of values - if that value exists in they jump table (an array of location to jump to) next assembly instruction will be to jump to that particular code block within the jump table. This removes the need for multiple comparison statements, and makes the assembly code easier to read. Switch statements will still evaluate the code blocks below the one jumped to, so it's best for the programmer to include a `break` after each code block within a switch statement.

API Patterns in Malware

Dynamic Linked Libraries (DLLs) are also a popular file type for malware authors. Unlike `.exe` files, `.dll` files have the ability export multiple functions, and are not runnable on their own. DLLs have no `entry point`, malicious `.dll` files usually have an exported function that is used as the entry point given a specific set of arguments. Submitting a `.dll` into a sandbox usually won't provide enough information to begin a more detailed analysis. We have to look further into `.dll` files to determine how they are used in an infection.

Viewing a `.dll` in IDA, we can see the identify `exports` of a `.dll` - the functions the `.dll` advertises for use. Most malware do not use the actual address of a function contained within a `.dll`, they use the **ordinal** value. Ordinal values are an integer reference to a specific function within a `.dll`. Malware authors commonly use these values to obfuscate their usage of the `.dll`, making it more challenging for an analyst to decipher the relevance of the function.

A **dropper** is a family of malware where the rest of the files required to conduct further infection of the target device is embedded within the executable. You can use these Windows API calls to begin fingerprinting droppers:

- FindResource
- LoadResource
- SizeofResource

- LockResource
- WriteFile
- CreateProcess
- CreateMutexA

We can extract embedded resources using PeStudio:

> select resources > dump (RAW)

This is useful because IDA does not disassemble the resources by default. IDA disassembles the executable before it runs, thus it will never see the outcome of the disassembled resource because it views the resource as just regular data.

Malware can also be used to monitor a user's activities. These are common Windows APIs used by malware authors to monitor keys, windows, and the clipboard:

- GetKeyState
- GetAsyncKeyState
- GetWindowText
- OpenClipboard
- GetClipboardData
- CloseClipboard

64-bit Code Analysis

32-bit malware is still the most prevalent, however, as 64-bit malware becomes more common, here are the two types that have been seen the most in this family:

- Browser Helper Objects for 64-bit Internet Explorer
- Device Drivers (rootkits) for Windows x64

64-bit Windows can still run 32-bit Windows applications, however, using the WoW64 subsystem (Windows on Windows). 32-bit applications can't leverage 64-bit DLLs, so they use 32-bit DLLs stored in %SystemRoot%\Syswow64. 32-bit applications also access the registry hive differently, using the 32-bit hive located under the registry "Wow6432Node".

Some differences exist when reading disassembled 64-bit code:

- All registers have been renamed (E** -> R**)
- There are eight (8) new general purpose registers (R8 - R15)
- RSP not (EBP or RBP) is used as the frame pointer for function calls. This is due to the fact that the stack size changes less frequently on 64-bit operating systems.
- The RIP (instruction pointer) can now be used to reference memory locations
- The common calling convention resembles fastcall - first four parameters for a function are passed in RCX, RDX, R8, and R9

Summary

This about wraps it up for code analysis after disassembling a malicious binary. To start code analysis, just remember these places and indicators for a good start:

- Imported functions
- Libraries
- Referenced strings
- Smaller functions called repeatedly
- Smaller functions with a few system calls
- Referenced resources

And always take advantage of previous behavioral analysis to lead the code analysis process. You'll know what you want to see from the code based upon the behavioral analysis. Then, the code analysis can provide you with further insight into the nature of the malware specimen.

610.2: Reversing Malicious Code

application data directory	common directory for malware to write to because access generally requires only user-level rights
attempt to connect to C2 server	malware that loops over a list of servers attempting to establish a connections
call instruction	an instruction that transfers control to the first instruction in a function
cdecl convention	most common function calling convention; the caller cleans up the stack
CloseClipboard	Windows API call to close the clipboard
control variable	variable(s) that are used to determine if a loop exists
CreateMutexA	Windows API commonly used by malware writers to signify that a device has already been infected; creates a mutex
CreateProcessW	Windows API call to create a new process; references to this function may reveal other processes spawned by a malware specimen
cs	default segment register when fetching instructions
data structure	refers to the layout and representation of information, and how we access and manipulate that representation
direct addressing	dereferencing the immediate value; usually annotated by disassemblers with brackets; ex. [0x410230]
dll	library file intended to share code with multiple programs; typically used to export functions
dropper	malware used to drop files embedded into the executable onto the target device
ds	default segment register for accessing data with ESI and EDI registers
dword	double word; 32-bits
eax	accumulator register; used for addition, multiplication, and return values
ebp - #	how to reference a local variable of a function using the frame pointer and an offset
ebp	register often used to reference arguments passed into a function as well as the local variables of a function; base pointer
ebp + #	how to reference a parameter passed into a function using the frame pointer and an offset
ebx / edx	generic registers used for various operations
ecx	counter register; commonly used for looping
effective address	the address of a data element, taking into account offsets due to array indexing and record accesses
eflags	status and control flags, each flag is a single binary bit
eip	instruction pointer; points to the next instruction to execute
encrypt / decrypt network traffic	malware that loops over each character in string before sending across the network
esi / edi	registers used for memory transfer functions
esp	stack pointer; used to point to the last item on the stack
exports tab	this tab in IDA displays the location of the entry point of the executable; for libraries or DLLs, this tab displays multiple exported functions and their location
fastcall convention	function calling convention where arguments are stored in registers; extra arguments are then placed on the stack; callee cleans up the stack
fast library identification and recognition technology (FLIRT)	technology used in IDA to automatically identify common libraries used by an executable
FindResourceW	Windows API call to determine the location of a resource
first operand addressing mode	register based addressing mode; using a register as an argument
function epilogue	occurs at the end of the function; cleans up the stack and restores registers
function prologue	occurs at the start of a function; allocates space for variables; saves registers that will be reused in the function body
GetAsyncKeyState	Windows API call to determine if a key is currently up or down, or if it was pressed since the last call to this API

36. **GetClipboardData** Windows API call to gather data from the clipboard; malware authors use this call to acquire usernames / passwords being copy / pasted

37. **GetKeyState** Windows API call to retrieve the status of a specified key

38. **GetTempFileNameW** Windows API function to create a name for a temporary file; malware often uses this API to name new files written to disk

39. **GetTempPathW** Windows API call often used by malware to create files names for temporary files on disk

40. **GetWindowText** Windows API call to obtain the text of a window's title bar

41. **ida graph view** pressing spacebar in IDA will present this view

42. **imports address table (IAT)** displays the APIs used by the program that are contained in external libraries

43. **inline function** function declared inline using the inline keyword or by being a member function defined in-class; removes overhead for entering or exiting the function; hard to determine the difference between inline functions and original code block

44. **ja** (unsigned) true if both carry and zero flag = 0

45. **jb** (unsigned) true if carry flag = 1

46. **je / jz** true if zero flag = 1

47. **jg** (signed) true if zero flag = 0 and sign flag = overflow flag

48. **jl** (signed) true if sign flag != overflow flag

49. **jmp** unconditionally jump to the label (address) in the operand

50. **jump table** a list of addresses of each code block; control is transferred to the desired block by using the variable to look up the address of the code block in the jump table; primarily used for compiling switch statements

51. **jz** jump if zero

52. **lea** load effective address into specified register

53. **leave instruction** mov esp, ebp
pop ebp

54. **linker** a program that combines the object program with other programs in the library, and is used in the program to create the executable code

55. **LockResource** Windows API call to obtain a pointer to a resource

56. **log keystrokes** malware that loops to check the state for each key code {0..92}

57. **loop body** code block that gets executed in a loop

58. **loop initialization** location where the starting value for a loop control variable is assigned (usually found outside the loop body)

59. **loop update** instructions that modify the control variables during each loop iteration

60. **object code** the output of the compiler, after translating the program

61. **OpenClipboard** Windows API call to get access to the clipboard and ensure other applications don't modify the clipboard data

62. **ordinal** alternative method to export and import functions; numerical value that can be used in place of a name; malware often exports or imports only this value to make it more challenging to decipher a function's relevance

63. **perform a port scan** malware that loops trying to connect to port 1- 65535

64. **performing a DDoS attack** malware that loops attempting to send a large amount of packets to a target host

65. **pointer** a variable that contains the address of some location in memory

66. **push instruction** an instruction usually used to push values to the stack for use in an API call

67. **qword** quadruple word; 64-bits

68. **ret** return to the calling function

69. **retn instruction** pop eip

70. **second operand addressing mode** memory address based addressing mode; using a memory address as an argument

71. **ShellExecuteW** Windows API call to facilitate command execution

72. **SizeofResource** Windows API call to obtain the size of a resource

73. **source code** human-readable code, not compiled

74.	**ss**	default segment register for accessing data wit the ESP register
75.	**stack**	typically used to store local variables in addition to parameters passed into a function
76.	**stdcall convention**	function calling convention used by WIN32 APIs; callee cleans up the stack
77.	**stopping conditions**	conditions used to determine if a loop should exit
78.	**strace**	monitors all the system calls made by a program
79.	**sysmon**	monitors system calls for registry and file-related activity
80.	**%systemroot%\Syswow64**	where 32-bit DLLs are stored for usage in the WoW64 subsystem
81.	**test instruction**	implied AND instruction; tests to see if a register, usually EAX, is zero
82.	**third operand addressing mode**	immediate based addressing mode; using the immediate value as an argument
83.	**thiscall convention**	function calling convention; used in C++ code member functions; convention includes a reference to "this" pointer; for Microsoft compilers, ECX holds the "this" reference - callee cleans up; for GNU compilers, "this" is pushed onto the stack last and the caller cleans up
84.	**word**	the natural size for a unit of data; currently taught to be 16-bits
85.	**WoW64**	acts as the emulator for allowing 32-bit applications to run seamlessly on a Windows 64-bit OS
86.	**WoW6432Node**	where the 32-bit compatible registry is located on a 64-bit Windows operating system

FOR610.3: Malicious Web and Document Files

Interacting with Malicious Sites and Infrastructure

In the past sections we've been examining malware isolated from the internet. Sometimes, however, in order to fully examine a specimen and its capabilities, we need to interact with the internet infrastructure that enables it.

Caution: you should attempt to conceal your identity and location as much as possible when researching malicious infrastructure. Malware authors might be tracking who visits their site and use your information to trace you back to your organization, or tag your IP address as an analyst attempting to determine the source of an infection.

When conducting OSINT on a target website, you might run into a webserver that is explicitly configured to determine if your browser is exploitable and attempt to infect your machine. If you would like to gain more information about this webserver, and coax the server to attempt and exploit, you could run a purposefully vulnerable browser in a lab environment and capture the network traffic.

Proxy options that exist in order to expose an interaction between your browser and the target webserver include:

- **Burpsuite**
- **Fiddler**

Alternatively, if you want to craft HTTP packets and spoof that you're using a browser to visit a website, these tools are available:

- **wget**
- **curl**
- **Pinpoint**
- **Scout**
- **Thug** (honeyclient)

Consider the following when investigating a compromised website suspected of hosting an exploit kit:

- What code as added to the compromised website?
- What hostnames or IPs were involved in attacking the visitors?
- What client-side software was likely targeted by the exploit?
- What type of malware was likely installed on victims' systems?

WMIC is commonly used by malware authors to spawn processes outside of the context of the current process they're exploiting. This allows the malware author to escape some limitations

that may exist for a child process that are imposed by a parent process.

You can carve out malicious files that were transferred in the exchange between a piece of malware and malicious infrastructure using these tools:

- **Wireshark** (Use File > Export Objects > HTTP)
- **CapTipper**
- **NetworkMiner**

Deobfuscating Scripts Using Debuggers

When encountering obfuscated Javascript, there are various methods to make the script human-readable again. Notepad++ contains a couple of features that will reformat and minimize unnecessary lines of code in a given piece of Javascript. These two tools are:

- **JSMin**
- **JSFormat**

There is also a Javascript beautifier available on REMnux called *js-beautify*. It can also be found at http://jsbeautifier.org.

Commonly used Javascript functions to execute malicious actions include:

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

There are methods for malware authors to defend themselves from being watched or deobfuscated during execution. `arguments.callee` is a javascript built-in that allows a function to reference itself. It's possible for a javascript function to attempt to detect if it has been modified - this allows malware authors to exit execution upon failing to pass their own checksums. `arguments.callee` can also be used as the decryption key for a function, any alterations will break the script. It's best to use debuggers that won't alter the script, and Internet Explorer provides a nice debugger that will place in-line breakpoints.

Deobfuscating Scripts Using Interpreters

In the previous section we utilized a browser's built-in debugging feature to run a script, set breakpoints, and step through the code at each point in its execution. We can also extract malicious scripts embedded in HTML or Adobe Reader files and run them in an interpreter specifically designed to execute Javascript. Some commonly used Javascript interpreters are:

- **SpiderMonkey** - Mozilla
- **CScript** - Internet Explorer

- **V8** - Google Chrome

Often when deobfuscating a script that was embedded within a browser, we need to redefine variables the script was attempting to use when within the context of the browser. With the tools listed above, it is possible to write a header file that redefines specific variables that the script is expecting, allowing the script to execute successfully. In REMnux there is an `objects.js` file that will define commonly used objects for browser based Javascript - allowing us to debug successfully if this file is included in a script's runtime.

It's best when downloading an embedded, malicious Javascript file that you save as much metadata as possible from the original HTML file. This way, you can provide the metadata the Javascript file is looking for to the interpreter. Sometimes the metadata of a web page is what the Javascript uses as keys to decrypt, etc.

Obfuscation of scripts involves trickery to confuse analysts and security tools. In summary, obfuscation has these attributes:

- **Unusual syntax of the code**
- **Generation of script elements on-the-fly**
- **External runtime dependencies**
- **Detection of script modifications**
- **Browser-specific implementations**

There are several more tools available than just these interpreters that we can use to deobfuscate malicious Javascript. Here's a list:

- **Kahu Security** - free tools designed to run on Windows for decoding content and deobfuscating malicious scripts
- **PhantomJS** - headless browser designed to run and debug Javascript
- **Nightmare** - another headless browser like PhantomJS
- **box-js** - Javascript engine that can emulate a browser or Windows runtime environment
- **malware-jail** - another Javascript engine like box-js

In summary:

- **Using standalone interpreters is sometimes faster or more convenient than using a debugger.**
- **You probably need to define objects for a malicious javascript excerpt in order to emulate a browser environment.**

Malicious PDF Document Analysis

PDF files are almost like HTML documents - well-structured, and you embedded different types of scripts into them that will be executed on a target device. In the lessons provided in FOR610, we extract embedded Powershell as well as Javascript from a PDF.

Different portions of a PDF are separated by `objects`. All text data, font info, or images are stored in `streams`. For our exercise in this section, the malicious document contained a Powershell script that was base64 encoded. It is possible to extract this manually by copy-pasting, however, there exist tools that can automatically parse a PDF, provide a listing of all its objects, and base64decode and output the contents of an encoded object. These tools include:

- **pdfid.py** - performs an initial quick assessment of a PDF file for suspicious keywords and dictionary entries
- **pdf-parser.py** - parses a PDF file, locates specific objects ad displays their contents
- **base64dump.py** - base64decode strings from PDF files
- **peepdf.py** - a good alternative to *pdfid.py* and *pdf-parser.py*

Often Javascript embedded into a PDF is used for heap spraying. At runtime, the script engine stores newly defined arrays on the heap. Malware authors declare each element of the array to be a copy of the shellcode. Thus, when the application is exploited, the instruction pointer can be pointed to a location in the heap that has a high likelihood of being the generated shellcode.

You might often find shellcode embedded into PDF files. The tools we already know how to use, IDA and x32/64dbg have the ability to interpret shellcode and provide the assembly instructions that they correspond to. A tool we can use to emulate shellcode is **scdbg**. **scdbg** expects shellcode in its raw, binary form and will provide the output of the shellcode in a GUI.

Sometimes **scdbg** fails to emulate shellcode properly. In these situations, you'll need to provide a stripped Windows executable for the shellcode to execute inside of. A useful tool to convert shellcode to a `.exe` is **shellcode2exe.py**.

PDF files could also be password protected to prevent analysis. Because of this, you'll be able to see the structure of the file, however, everything will be encrypted - you'll have to supply a password to decrypt the contents. If you know the password, there are multiple CLI programs you can use to decrypt a PDF:

- **qpdf**
- **pdftk**

Another complication of PDF analysis is an object that contains a stream in its dictionary, and that stream embeds other objects. **Object streams (/ObjStrm)**, as they're called, can be located and parse by **pdf-parser.py**.

Other useful tools for PDF analysis:

- **swf_mastah.py** - extracting Flash from PDF files
- **Origami PDF Framework**
- **PDF Stream Dumper**

In summary, PDF analysis can be wrapped into these salient points:

- **Look for risky or otherwise unusual objects.**
- **Locate, extract, and decode code that would execute on the victim's system.**
- **Several tools exist for analysis and extraction of embedded objects in malicious PDF files.**

Macros in Malicious Office Documents

Microsoft Office documents are a very common way to spread malware as they are the most commonly used document file in a business. Microsoft Office documents allow adversaries to embed macros in a file. The macros are written in Visual Basic for Applications (VBA) - a language that supports powerful capabilities for interacting with the system.

olevba.py is tool that can be used to extract VBA macros from Microsoft Office documents without relying on the Microsoft Office software suite. **olevba.py** can automatically parse contents of Microsoft Office files, extract, and display any embedded macros.

Something to note about Microsoft Office documents - there are two document formats:

- **OLE2** - Object Linking and Embedding 2; legacy version - sometimes called Structured Storage (SS) or Compound File Binary Format (CFBF)
- **OOXML** - Office Open XML; well formatted and easier to read - less likely to contain vulnerabilities; all file extension end in `m`:
  - `.docm`
  - `.xlsm`
  - `.pptm`
  - `.dotm`

Tools that enable you to examine the structure of OLE2 files are:

- **oledump.py**
- **olecfinfo**
- **oledir.py**
- **olebrowse.py**
- **SSview**

Malware authors often obfuscate their malicious VBA scripts using the built-in function `XORI` to decode the script at runtime. **xor-kpa.py** is a tool that can derive a XOR key from a ciphertext given a piece of plaintext contained within the ciphertext.

oledump.py contains a plugin, **plugin_http_heuristics**, that will automatically locate URLs embedded within obfuscated OLE2 files if the malware author is using common obfuscation methods.

Sometimes dynamic analysis of VBA scripts is easier than static analysis, especially if the macro is heavily obfuscated. You can do this by creating a new Microsoft Office document, acquiring the

VBA script embedded in the original malicious Microsoft Office document, and copy / pasting the malicious VBA script into the macro editor of the new document. From here, you can set breakpoints at different sections of the VBA code, allowing you to stop before it executes completely. You can view all of the local variables and watch them change as you step through the code.

All the tools mentioned previously search for VBA macro source code, however, before executing VBA macros, Microsoft Office compiles VBA macros into **p-code**. Theoretically, a malware author could generate **p-code** and embed it within a Microsoft Office document - the scanners mentioned previously would never detect it. Luckily, we have a tool called **pcodedump.py** that will locate and disassemble **p-code** for our analysis.

In summary:

- **VBA macros provide attackers with a convenient and powerful way to execute malicious code on victims' systems**
- **Macros can interact with the network, file system, and other aspects of the environment.**
- **Macros are embedded in OLE2 binary files and are supported by all Microsoft Office versions in use today.**
- **Some macros plainly reveal their functionality, others employ obfuscation or trickery.**

Malicious RTF Documents

Rich Text File (RTF) is a document format designed by Microsft and is a "method of encoding formatted text and graphics for use within applications and for transfer between applications". Malicious **RTF** files are written for Microsoft Word and, while they don't allow for the embedding of macros, **RTF** files allow for arbitrary files to be embedded in **RTF** documents as objects using version 1 of the OLE formate (OLE1) - sometimes referred to as the Package Object Server.

Malware authors take advantage of how Microsoft Word handles objects embedded in **RTF** files. When Word opens **RTF** documents, it automatically extracts any embedded objects and stores them in the `%Temp%` folder. From here, a malware author can embed a macro that will execute the file stored in `%Temp%`. Malware authors can effectively **RTF** files to act as containers for malicious code.

RTF files written by malware authors will usually contain `\objects` with `\objdata`. This data is encoded, however, we can use a tool call **rtfdump.py** that can parse through **RTF** documents and extract embedded objects.

In summary:

- **RTF documents are convenient carriers of other malicious files.**
- **Look for embedded objects and anomalous content when assessing and RTF file.**

- **Be prepared to locate, extract, and analyze shellcode:**
    - Try emulating its executing for API-level visibility
    - Analyze its code with debuggers and disassemblers
    - Observe its effects using behavioral monitoring tools

610.3: Malicious Web and Document Files

1. **/AcroForm** — pdf object designed to embed interactive forms in pdf files; used by malicious authors of pdf files
2. **\aftnrestart** — rtf control word restarts endnote numbering each section
3. **appendChild** — this method appends a node as the last child of a node; javascript built-in; commonly used for malware obfuscation
4. **app.setTimeOut** — javascript embedded in a PDF trick; indirect way of launching a designated function instead of executing directly using eval; can be used to delay execution until the document is fully loaded
5. **app.viewerVersion** — javascript embedded in a PDF; used to identify the version of the PDF viewer being used
6. **arguments.callee** — javascript attributes some malware authors use to reference the javascript code itself; malware authors will attempt to detect changes to the javascript code in order to protect themselves being watched during execution
7. **AutoOpen** — a macro that runs when opening a document that contains the macro
8. **beautification** — reformatting malicious scripts in order to read them easier; this is the first step in deobfuscating malicious scripts
9. **box-js** — javascript interpreter that will deobfuscate and analyze malicious javascript; provides a listing of all URLs the script attempts to connect to; can emulate browser environments
10. **CapTipper** — specialized HTTP analysis tool written in Python that will analyze a .pcap file and carve all files transferred via HTTP
11. **curl** — like wget; allows user to craft their HTTP request
12. **debugger** — this is a keyword that can be used in Internet Explorer to set a breakpoint in the middle of a script
13. **document.getElementById** — returns the element that has the ID attribute with the specified value from an HTML document; used by malware authors to create dependencies within the Javascript for specific HTML elements
14. **-EncodedCommand** — powershell option that specifies that a specific command is base64 encoded
15. **eval** — this function evaluates or executes an argument ; javascript built-in ;commonly used for malware obfuscation
16. **fileinsight** — lightweight hex editor that has many capabilities and plugins useful for malware analysis
17. **fs:[0x30]** — location of the pointer to the process environment block in every thread information block
18. **generation number** — in pdf object specification, this is the second number of an object's definition
19. **geteip** — a technique involving making a `call` instruction in order to have the eip pushed onto the stack, and then immediately calling pop in order to acquire the value of eip
20. **GoTo** — VBA branching instruction used to obfuscate and confuse analysts
21. **headless browser** — browsers useful for deobfuscating scripts; less-specialized and stripped down browsers that are primarily used for malware analysis
22. **heap spraying** — placing shellcode in numerous locations of a program's heap memory so that, when an exploit occurs, no matter where the instruction pointer lands it will execute the shellcode
23. **honeypot** — decoy servers or systems setup to gather information regarding an attacker or intruder into your system

24. **iframe** tag that represents an inline-frame; commonly used to inject malicious code into a web response;
25. **indirect object** pdf objects that have a unique identifier and can be referenced by other objects
26. **insertBefore** this method inserts a node as a child, right before an existing child, which you specify; javascript built-in; commonly used for malware obfuscation
27. **/JavaScript** pdf keyword that usually is a strong indicator the pdf is malicious
28. **jmp2it** rather than generating an executable out of shellcode like shellcode2.exe, this tool directly executes the shellcode located in a specified file
29. **js-beautify** javascript beautifier available on REMnux
30. **JSFormat** notepad++ feature; inserts line breaks and indentations to make Javascript easier to read
31. **JSMin** notepad++ feature to get rid of extraneous Javascript components such as comments
32. **/Launch** pdf keyword to execute a specified file
33. **location.href** javascript built-in to reference the URL of the web page
34. **NetworkMiner** another specialized network analysis tool that can extract files from HTTP sessions
35. **\objdata** rtf control word containing an object's object data
36. **\object** rtf control word that specifies an object and its data follows
37. **object number** in pdf object specification, this is the first number of an object's definition
38. **/ObjStrm** a pdf stream object that contains a stream of other embedded objects; can be used to obfuscate and confuse analysts
39. **OLE2** object linking and embedding 2; legacy Microsoft Office file format; sometimes called Structured Storage (SS) and Compound File Binary Format (CFBF)
40. **oledump.py** CLI tool that allows you to examine the structure of OLE2 files
41. **olevba.py** CLI utility that can automatically parse contents of Microsoft Office files, extract, and display embedded macros
42. **OOXML** Office Open XML; Microsoft Office file format; easier to parse and less likely to have vulnerabilities
43. **/OpenAction** pdf keyword that specifies what action an application will take after opening a file
44. **p-code** this type of bytecode is generated when Microsoft Office compiles VBA macro source code
45. **pcodedmp.py** CLI tool that can locate and extract VBA macro p-code embedded in Microsoft Office documents
46. **pdftk** another CLI utility that can decrypt PDF files given the correct password; doesn't work will if a PDF file contains malformed objects
47. **Pinpoint** fetches a webpage and then enumerates and analyzes its components to help identify any infected files.; gives you various options when making an HTTP request including spoofing the user-agent string and referrer; will not render any of the content
48. **process environment block** Windows operating system data structure that contains information about a process including the list of its DLLs that have been loaded or mapping into the process's memory
49. **qpdf** CLI utility that can decrypt PDF files given the correct password
50. **regular expressions** the special metacharacters used to match patterns of text within text files; commonly used by malware authors to obfuscate strings
51. **/Root** pdf keyword for a document's root
52. **rtf** rich text file; allows formatting of text and inserting graphics; used by malware authors to embed malicious objects in Word documents - usually files
53. **rtfdump.py** cli tool to parse through RTF files and extract embedded objects
54. **scdbg** shellcode emulator; expects shellcode in its raw binary form
55. **Scout** uses the Pinpoint engine to download and analyze webpage components to identify infected files; works fine in 32-bit Windows; has a built-in HTTP Request Simulator that will render user-specified HTML files, catch the resulting HTTP requests, then drop the responses; includes the ability to screenshot the webpage using PhantomJS (download PhantomJS and copy the .exe to the same folder)
56. **streams** pdf method of storing data such as text, font definitions, and pictures

57.	**swf_mastah.py**	CLI utility that can extract Flash objects from PDF files
58.	**syncAnnotScan / getAnnots**	javascript embedded in a PDF; used to enable the script to store some of its contents as annotations to allow it to assemble itself on runtime
59.	**ternary operator**	used for one-line conditional statements; used to obfuscate code and confuse readers
60.	**thread information block**	windows operating system data structure that contains information about the currently running thread
61.	**Thug**	python low-interaction honeyclient
62.	**tuple**	a type in Javascript that can contain multiple different values of different types; Javascript only assigns the last element to reference the variable
63.	**vbaProject.bin**	the default name for macros store in Microsoft Office XML-documents
64.	**wget**	non-interactive network downloader
65.	**\windowcaption**	rtf control word used to set the caption text for the document window
66.	**wmic**	Windows Management Instrumentation Command Line Tool; commonly used by malware authors to escape restrictions / limitations imposed by a parent process; spawns a completely new process under wininit
67.	**Workbook_Open**	a macro that run when opening a spreadsheet that contains the macro
68.	**write**	this method writes HTML expressions or JavaScript code to a document; javascript built-in; commonly used for malware obfuscation
69.	**/XFA**	another pdf object designed to embed interactive forms in pdf files for malicious purposes; XML Forms Architecture
70.	**XORI**	VBA built-in function that can be used to XOR two strings together
71.	**xor-kpa.py**	CLI tool to automatically derive a XOR key by examining the plaintext and the ciphertext that contains the encoded version of that plaintext
72.	**xorsearch**	cli tool designed to search a specified file for the presence of a specified string encoded using common obfuscation techniques; can also be used to discover shellcode patterns

FOR610.4: In-Depth Malware Analysis

Recognizing Packed Malware

Malware authors use tools called *packers* to protect their creations from anti-malware products and analyst tools. We, as malware analysts, need to understand how these *packers* work and be prepared to examine packed malware specimens.

Here's a list of commonly used packers:

- **UPX**
- **Armadillo**
- **FSG**
- **Themida**

To detect if a malware specimen has been packed during your initial static analysis, look for these common identifiers:A good way to detect if a malware specimen is packed is:

- If you see very few readable strings
- If your disassembler recognizes very few functions within the program
- If your disassembler recognizes very few API calls within the program
- If the entropy of the file is too high
- Packer based signatures exist in the file

Tools that can be used to detect packed executables:

- **Bytehist**
- **pescanner.py**
- **Detect It Easy**
- **Exeinfo PE**
- **trid**
- **pepack**
- **packerid**
- **pescan**
- **ProtectionID**
- **RDG Packer Detector**
- **CFF Explorer**

Getting Started with Unpacking

UPX can usually unpack malware that has been previously identified to be packed, however, there are other tools available:

- **TitanMist**
- **Ether**

Sometimes we won't be able to unpack and extract malware with these automatic tools and, to do our jobs effectively, we'll have to conduct all of our analysis manually.

One obstacle standing in our way is **ASLR** (address space layout randomization). This is a feature for operating systems that allows operating systems to ignore an executable's base address and randomizes address locations. This prevents hackers from being able to determine the location of different resources within an executable, increasing security.

Below are two tools that can be used to disable ASLR for portable executables:

- **CFF Explorer**
- **setdllcharacteristics**

Disabling **ASLR** will ease the difficulty of our analysis, allowing us the ability to track down the location of the unpacking code and the beginning of the unpacked, malicious executable.

So how do we go about acquire the malicious code that has been packed? We conduct a process call **dumping**, allowing the unpacker to load the malicious executable into memory and then using a tool to dump the running process into a file on disk.

Often the dumped file might be broken when we attempt to run it - probably because the entry point of the PE is pointing to the unpacker code, but we need to begin at the unpacked code. Usually the import address table (IAT) is also mangled, and the executable doesn't know how to locate its resources.

Here are some tools aimed at **dumping** unpacked executables from memory to disk, as well as reconstructing an executable's entry point and IAT:

- **Scylla**
- **PE Tools**
- **Universal Import Fixer**
- **Imports Fixer**

In summary, to begin unpacking malware, here are some important steps and things to remember:

- **Disable ASLR on packed programs in order to make analysis easier.**
- **Allow the malicious program to unpack itself; then dump it.**
- **Dumped files might not be runnable because the entry point is broken.**

Using Debuggers for Dumping

Using debuggers to unpack and extract packed executables is a safer and more precise way of

acquiring packed malicious code. In order to do this, we need to set a breakpoint at the end of the unpacking code in the debugger. This is usually a JMP or CALL instruction pointing to the unpacked code's Original Entry Point (OEP). You can also identify the ending of unpacked code by looking for a location filled with lots of zeros and no instructions remaining after that.

After reaching unpacked code, in a debugger like x64dbg, we can search for newly existing strings and intermodular calls to confirm that we have found the unpacked code. In most debuggers, you can just right-click the assembly you're looking at and then search for these things.

Debugging Packed Malware

Sometimes it's best for us to analyze the packed malware within a debugger and we don't want to extract the code. In order to do this, we should let the malware run without any breakpoints. View the malware's memory regions with the debugger, and search for memory regions that have the "execute" flag set - this denotes these memory regions are allowed to execute instructions on the CPU. Navigate to that particular memory region and search for interesting strings or API calls (intermodular calls) like we did in the previous section. This will provide you with locations of interest within the packed code that you can set **hardware breakpoints** at. We want to set **hardware breakpoints** because those are less likely to be ignored than software breakpoints upon restarting the process.

After setting our **hardware breakpoint**, we will proceed to debug the code and run until we stop at the breakpoint. From there, we should be stopped within the unpacked code. This will allow us to analyze the unpacked code without extracting it from the process.

Code Injection and API Hooking

Malware authors utilize code injection to hide extracted code into other processes. This makes it harder for incident responders and analysts to locate malicious code. Malware also uses code injection to implement **rootkits.** These user-mode rootkits hook into system APIs to interfere with the normal flow of information within the infected process.

Here are some common Windows API calls used by malware authors to inject code into processes:

- **CreateToolhelp32Snapshot**
- **Process32First**
- **Process32Next**
- **EnumProcess**
- **OpenProcess**
- **CreateProcess**
- **WriteProcessMemory**
- **CreateRemoteThread**

- **GetModuleHandle**
- **GetProcAddress**
- **CreateRemoteThread**

A common method of injecting code into another process is:

Createtoolhelp32Snapshot -> Find Process Handle -> OpenProcess -> VirtualAllocEx -> WriteProcessMemory -> CreateRemoteThread

Another method is to force another process to load a malicious .dll:

OpenProcess -> VirtualAllocEx -> WriteProcessMemory (write .dll location) -> GetModuleHandle (find kernel32.dll) -> GetProcAddress (find LoadLibrary) -> CreateRemoteThread (execute LoadLibrary with .dll location as an arguemnt)

User-mode rootkits usually use these API calls:

- **ReadProcessMemory**
- **VirtualProtect**
- **WriteProcessMemory**

Malware Memory Forensics

Essentially this section is about conduct malware analysis on the memory image of an infected system. Memory forensics can supplement code and behavioral analysis, and allows us to identify forensically significant artifacts related to the host's active processes, their code and data, network connections, open files, registry contents, etc.

Software utilized for capturing memory images includes:

- **WinPMEM**
- **Comae Memory Toolkit (DumpIt)**
- **KnTDD**
- **BelkaSoft Live RAM Capturer**

Other possible methods of capturing memory images including utilizing specialized hardware tools. IEEE 1394 standard allows for direct memory access from FireWire devices, allowing us to acquire a memory image without host OS intervention. It's also possible to capture the system's hibernation file and convert it to a useable format for memory forensics tools.

If we have an infected virtual machine, we can just capture a snapshot of the virtual machine and analyze its memory from there.

Here are a list of popular memory forensics tools:

- **Volatility Framework**
- **Rekall**

- **Redline**

610.4: In-Depth Malware Analysis

Study online at quizlet.com/_7tubtr

1. **apihooks** — volatility module to detect with processes and DLLs have been modified with inline hooks
2. **ASLR** — Address Space Layout Randomization
3. **call table hook** — replaces the address of the targeted function in a table that processes use to find the function with the location of a rootkit function
4. **cleardb** — x64dbg command to delete all analysis details
5. **cmdline** — volatility module that will display to command line command used to invoke all processes
6. **CreateProcess** — Windows API call used to create another process
7. **CreateRemoteThread** — Windows API call to execute injected code inside a targeted process
8. **CreateToolhelp32Snapshot** — Windows API call used to get a listing of the currently running processes
9. **detect it easy** — signature-based scanner that attempts to identify packed malware samples
10. **dlllist** — volatility module to list the DLLs loaded into every process on the infected host
11. **dumping** — extracting an unpacked program from an infected host's memory
12. **dynamicbase flag** — a flag located in a PE's DllCharacteristics field that determines whether or not a PE supports ASLR
13. **entropy** — you can use this characteristic
14. **EnumProcess** — Windows API call similar to CreateToolhelp32Snapshot
15. **ether** — web based tool that attempts to automatically unpack malware specimens
16. **execute flag** — specific flag set for portions of memory of a process; used to track down unpack parts of code in a process's memory
17. **exeinfo PE** — another signature-based scanner like 'detect it easy' that attempts to identify packed malware samples
18. **hardware breakpoint** — breakpoint that is more likely to remain after the reloading of process; tied specifically to a memory register
19. **hooking** — intercepting system-level function calls, events, or messages
20. **impscan** — volatility module to examine a process in a memory image and extract API name and address information from it
21. **inline hook** — involves patching the beginning of targeted functions in memory of a compromised process; forces process to execute a rootkit
22. **intermodular calls** — typically equivalent to API calls; usually revealed when code is unpacked successfully
23. **kdbgscan** — volatility module that attempts to detect the OS profile
24. **ldrmodules** — another volatility module used to detect DLLs loaded into processes
25. **malfind** — volatility module used to detect concealed, injected code in a memory image
26. **memdump** — volatility module that will dump the contents of a process from a memory image to a file
27. **mshta.exe** — a program built into Windows for executing HTML applications
28. **OllyDumpEx** — useful plugin for x32/64dbg that allows a user to dump the process currently being debugged
29. **OpenProcess** — Windows API call used to open a process using its handle
30. **original entry point** — the original location where the packed code begins execution; the end of unpacking code JMPs or CALLs this location
31. **packers** — tools that compress, obfuscate, encrypt or otherwise encode the malicious code
32. **pescanner.py** — portable executable scanner that can detect and flag entropy of packed malware
33. **Powershell ISE** — an integrated scripting environment that includes a text editor.
34. **Process32First** — Windows API call used to start at a listing generated by CreateToolhelp32Snapshot
35. **Process32Next** — Windows API call used to iterate through a listing generated by CreateToolhelp32Snapshot

36. **ReadProcessMemory**	Windows API call used to read the first few bytes of a targeted function; allows a rootkit to save function locations for future use
37. **reg_export**	command line tool that is useful for extracting data from registry keys
38. **rootkit**	software that can conceal malicious artifacts from the user of the infected system
39. **scylla**	tool used to reconstruct the entry point and import address table of an unpacked, dumped malware specimen
40. **titanmist**	powerful framework for implementing your own unpack
41. **upx**	open-source portable executable packer
42. **VirtualAlloc**	Windows API call that allows a process to allocate, clear, and retrieve a pointer to space within the process's memory
43. **VirtualAllocEx**	Windows API call that allows a process to allocate, clear, and retrieve a pointer to space within another process's memory
44. **VirtualProtect**	Windows API call used to modify permissions on a targeted memory region to make sure it is writeable
45. **volatility**	free, popular, and powerful toolkit for conducting memory forensics
46. **WriteAllBytes**	powershell ISE command to save bytes of a variable to a file
47. **WriteProcessMemory**	Windows API call to write specified contents to a designated memory area

FOR610.5: Examining Self-Defending Malware

Debugger Detection and Data Protection

Malware that's attempting to evade analysis is obviously going to work to avoid being debugged by a malware analyst. For Windows executables, here are some common API calls used by malware to detect if it's being debugged:

- **IsDebuggerPresent**
- **CheckRemoteDebuggerPresent**
- **NtQueryInformationProcess**
- **ZwQueryInformationProcess**
- **OutputDebugString**

All of these API calls can be fooled by debuggers today, including x64/32dbg, by masking the expected response for the API calls to fool the process into assessing that it isn't being debugged. With all that said, be on the watch out for malware that attempts to check its **Process Execution Block (PEB)** directly, located at **FS:[30h]**. There is a 1-bit field called **BeingDebugged** that will identify that a process is currently being debugged, and malware authors attempt to check this rather than using the APIs listed above.

Some malware attempts to also conduct time analysis in order to determine if its running too slowly. Here are the common API calls that malware uses to detect it's being ran slowly within a debugger:

- **GetTickCount**
- **GetLocalTime**
- **GetSystemTime**
- **NtQuerySystemTime**

Malware can also use the assembly instruction **RDTSC (Read Time-Stamp Counter)** to determine how many ticks have passed since the system booted up. This can be used to access hardware values in order to avoid using the API calls above for time analysis.

Changing gears, let's talk about string obfuscation. Malware authors want to protect the strings they use from malware analysts as un-obfuscated strings can provide an analyst information that reveals the capabilities of the malware specimen. Here are some tools that can automatically detect the obfuscation method used to obfuscate strings within a malicious binary:

- **XORSearch**
- **brxor.py**
- **brutexor.py**
- **bbcrack.py**

- **xorBruteForcer.py**
- **NoMoreXOR.py**
- **xortool**
- **unXOR**
- **Kahu tools**

Another technique malware authors use to obfuscate strings are by creating **stack strings.** Malware authors will create an array of characters of the string they intend to use, and then build the final string in a buffer at runtime. This prevents string analyzers from finding the strings in the final binary, and also keeps strings that are intended to be used out of the **.data** section of the portable executable.

Tools that are able to un-obfuscate stack strings are:

- **stdeob.pl**
- **FLOSS (FireEye Labs Obfuscated Strings Solver)**

Unpacking Process Hollowing

Process hollowing is when malware launches a process in a suspended state, deallocates the memory containing that process's code, and replaces the process with the code If a malicious program.

Here are a list of Windows API calls used by malware authors to conduct **process hollowing**:

- **Create Process**
- **NtUnmapViewOfSection**
- **ZwUnmapViewOfSection**
- **WriteProcessMemory**
- **ResumeThread**

Malware authors often use **process hollowing** in order to conceal malicious code in what would normally look like a legitimate process.

In this section we learned:

- **How process hollowing works and the Windows API calls involved.**
- **How to debug a malware specimen that attempts to conduct process hollowing and dumping its unpacked code to a file.**
- **How malware authors conceal API calls and avoid including them the Import Address Table (IAT).**

Detecting the Analysis Toolkit

Malware authors obviously want to protect their malware from being analyzed and reverse

engineered. Here are a couple of methods, summarized, that malware authors use to detect their environment to determine if it's worth infecting.

- **Looking for Windows applications that end users will have installed.**
- **Looking for signs that applications like Wireshark, Process Hacker, or IDA are installed.**
- **Looking for specific hardware components to detect if the environment is virtualized.**
- **Looking for registry keys associated with VMWare Tools.**
- **Looking for**:
    - Contents in the clipboard.
    - Number of CPU cores.
    - Is the mouse cursor moving?
    - Is the hard disk reasonably large?
    - Does the `uptime` make sense?

Malware authors also try to trigger on user-interaction events to detect if the malware is within a sandbox. An example is using hooks to capture mouse interaction with a window, attempting to detect the press and depress during a click. This is accomplished by using the Windows API: **SetWindowsHookExA**.

Handling Misdirection Techniques

Malware can attempt to misdirect us by throwing an exception in order to hide its true entry point. The **Structured Exception Handling**, provided in portable executables, allows a programmer to define exception handling functions for a program. There are two types of SEH:

```
* Frame-based exception handling (32-bit programs; implemented in a link
* Table-based exception handling (64-bit programs; compiler creates a ta
```

It's fairly simple to track down the newly installed exception handler and watch it unpack the code, however, there are other methods. **Thread Local Storage (TLS)** callback functions allow the malware author to execute code before the program starts. **TLS** callbacks allow malware authors to create code that will be executed even before the Entry Point. Debuggers usually automatically execute **TLS** callbacks before pausing at the Entry Point, leading to premature execute of the malicious code.

Quizlet 610.5: Examining Self-Defending Malware

Study online at quizlet.com/_7uem7l

1. **0xcc**	hex value that corresponds to opcode INT 3; malware authors check for this in order to determine if someone set a software breakpoint
2. **BlockInput**	Windows API call used to block input
3. **CreateProcessA**	Windows API call that allows a program to launch another process
4. **FindWindow**	Windows API call used to find windows that are open; used by malware authors to detect if debuggers are open.
5. **Frame-based SEH**	keeps track of exception-handling records (structures) using a linked list called the SEH chain; hosted at FS:[0]
6. **GetCursorPos**	Windows API call to determine if the mouse has moved recently; malware authors use this to see if the host they're on is real
7. **GetModuleHandleW**	Windows API call to locate a handle to a .dll; used by malware authors to detect the existence of anti-virus software
8. **GetProcAddress**	Windows API call to get the procedure address of a function exported from a DLL
9. **IsDebuggerPresent**	Windows API call used to read the PEB to determine if the debugger bit is set
10. **KdDebuggerEnabled**	Windows API call to check for the existence of a kernel debugger
11. **LoadLibraryW**	Windows API call to load a DLL into the process space
12. **NtUnmapViewOfSection**	Windows API call to deallocate virtual memory of a process
13. **pe_unmapper**	tool used to post-process file dumps to make rebasing and conduct other tweaks of a PE's virtual to physical memory mapping
14. **process hollowing**	malware launches a process in a suspended state, deallocates the memory containing that process's code, and replaces the process with the code of a malicious program
15. **RegOpenKeyExW**	Windows API call used to open registry keys for reading; malware authors utilize this to detect the existence of virtual machine registry keys
16. **RtlDecompressBuffer**	Windows API to decompress data within a process's buffer; can be used to unpack code
17. **scyllahide**	x64/32dbg plugin that allows a malware analysis to cloak the presence of a debugger from the malware specimen
18. **SetWindowsHookExA**	Windows API call to hook user interaction with Windows; malware authors use this to detect a sandbox environment
19. **stack strings**	the storage of a string within a program in an array of characters, making it harder to piece together to final string
20. **strdeob.pl**	tool that disassembles a malware specimen and attempts to rebuild stack strings
21. **Structured Exception Handling**	a mechanism for graciously handling errors; malware authors abuse this to misdirect the analyst
22. **thread local storage (TLS)**	allows each thread to have its own copy of data; allows malware authors to execute callback functions before the debugger reaches the Entry Point
23. **VirtualProtect**	Windows API call used to modify the permission of a memory page; used by malware authors after unpacking code to prepare a page for execution