Malware

Foundations of Cybersecurity

Luca Caviglione

Institute for Applied Mathematics and Information Technologies

National Research Council of Italy

luca.caviglione@cnr.it

University of Pavia – Department of Electrical, Computer and Biomedical Engineering

Outline

- Why security malware in this course?
- What is malware?
- Malware Architecture
- Malware Analysis
- Static Analysis
- Packers
- Dynamic Analysis
- Anti-forensics
- YARA Rules
- Binary Obfuscation
- Source Code Obfuscation
- JavaScript Minification

Why Malware in this Course?

- Malicious software is one of the biggest headache for security guys.
- Offensive campaigns target:
  - individuals
  - states
  - corporations
  - critical infrastructures.
- Studying some internals of malware:
  - allows to introduce several advanced techniques
  - requires to dig various tools
  - is challenging and requires to grasp some advanced concepts
  - should be part of the knowledge of any engineer using hardware/software tools.

What is Malware?

- **Malware** (**mal**icious soft**ware**) is an umbrella term for intrusive software developed by cybercriminals for:
  - stealing data or resources
  - inflicting a damage to an asset
  - spying individuals or organizations
  - gaining control over a system
  - extorting moneys.
- Some major goals of malware campaigns:
  - industrial espionage
  - collect ransoms
  - send spam emails
  - assimilate nodes to botnets (e.g., for DDoS attacks).

What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Virus**: software able to copy and propagate itself, usually by injecting its code in other executable artifacts
  - **Worm**: a stand-alone software that can propagate itself autonomously

What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Virus**: software able to copy and propagate itself, usually by injecting its code in other executable artifacts
  - **Worm**: a stand-alone software that can propagate itself autonomously

Virus and **Worm**: in general, the **virus** requires an **intervention** from the user (e.g., to execute the infected file), whereas the **worm** can spread **without infecting** files.

What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Virus**: software able to copy and propagate itself, usually by injecting its code in other executable artifacts
  - **Worm**: a stand-alone software that can propagate itself autonomously
  - **Trojan**: software that masks itself as an innocent program for tricking users to install it. The trojan can then steal data, upload files or monitor users. Usually spread via social engineering or drive-by-download scheme
  - **Backdoor** or **Remote Access Trojan (RAT)**: software allowing the attacker to gain access and execute commands over the infected system

What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Dropper** or **Downloader**: software designed with the solely purpose of downloading or installing additional components, e.g., attack routines. They are typically lightweight with the goal of reduce the chance of detection
  - **Botnet**: nodes infected by the same software that can be orchestrated to generate traffic against a target or deliver spam
  - **Rootkit**: software designed to grant the attacker a privileged access to an infected system while concealing its presence

What is Malware?

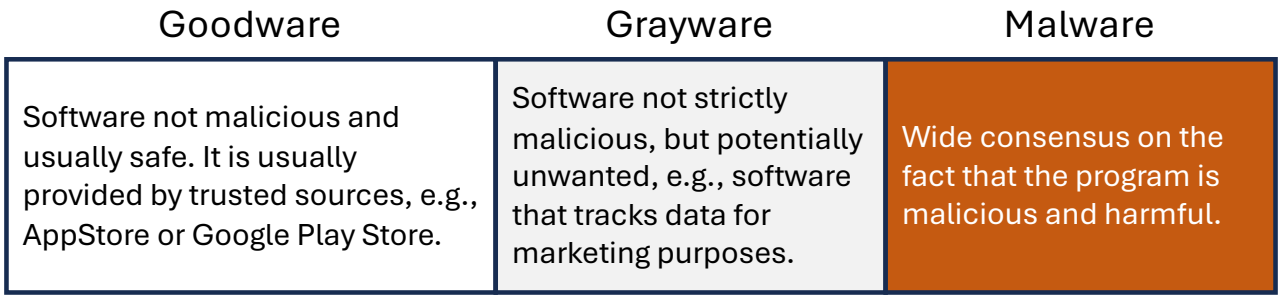
- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Information Stealer**: software designed to exfiltrate sensitive data, such as banking credentials. Typical examples are keyloggers, sniffers, spywares and form grabbers
  - **Click Fraud**: software aimed at acting as a real user clicking over a banner or an advertising link
  - **Ransomware**: software encrypting data of users or preventing the access to a system in change of money (cryptocurrencies)

What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Adware**: software showing unwanted advertisements or profiling users to support advertising campaigns
  - **Cryptominers**: software designed to steal resources of the infected host (e.g., CPU and bandwidth) for mining cryptocurrencies.

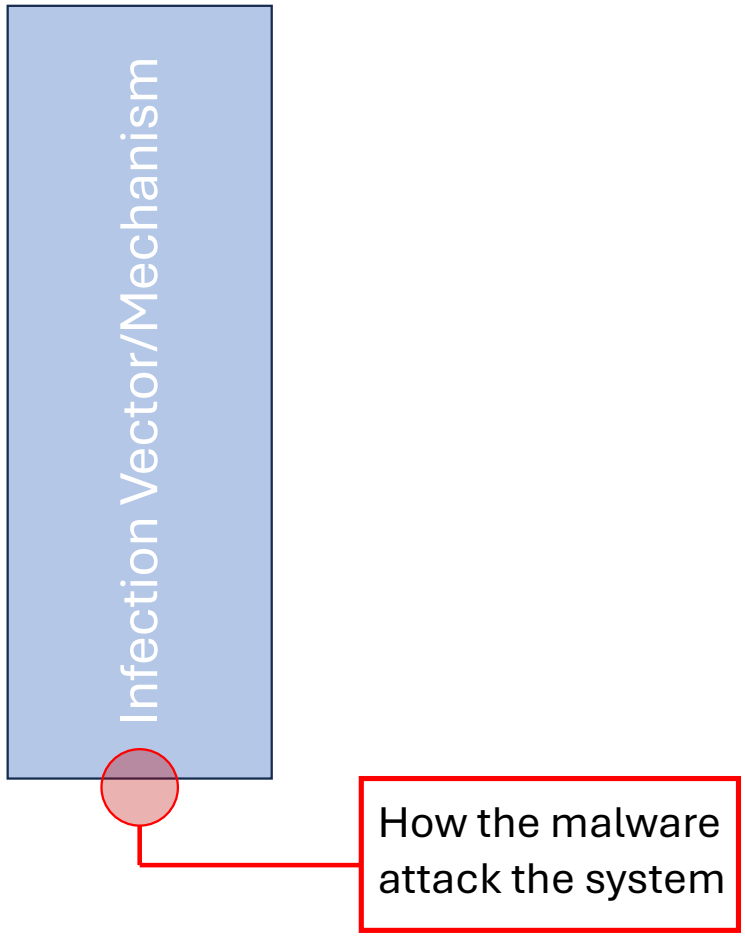
What is Malware?

- Malicious programs are often (broadly) **categorized** according to:
  - **functionalities**
  - **attack vectors**.
- Main types are:
  - **Adware**: software showing unwanted advertisements or profiling users to support advertising campaigns
  - **Cryptominers**: software designed to steal resources of the infected host (e.g., CPU and bandwidth) for mining cryptocurrencies.

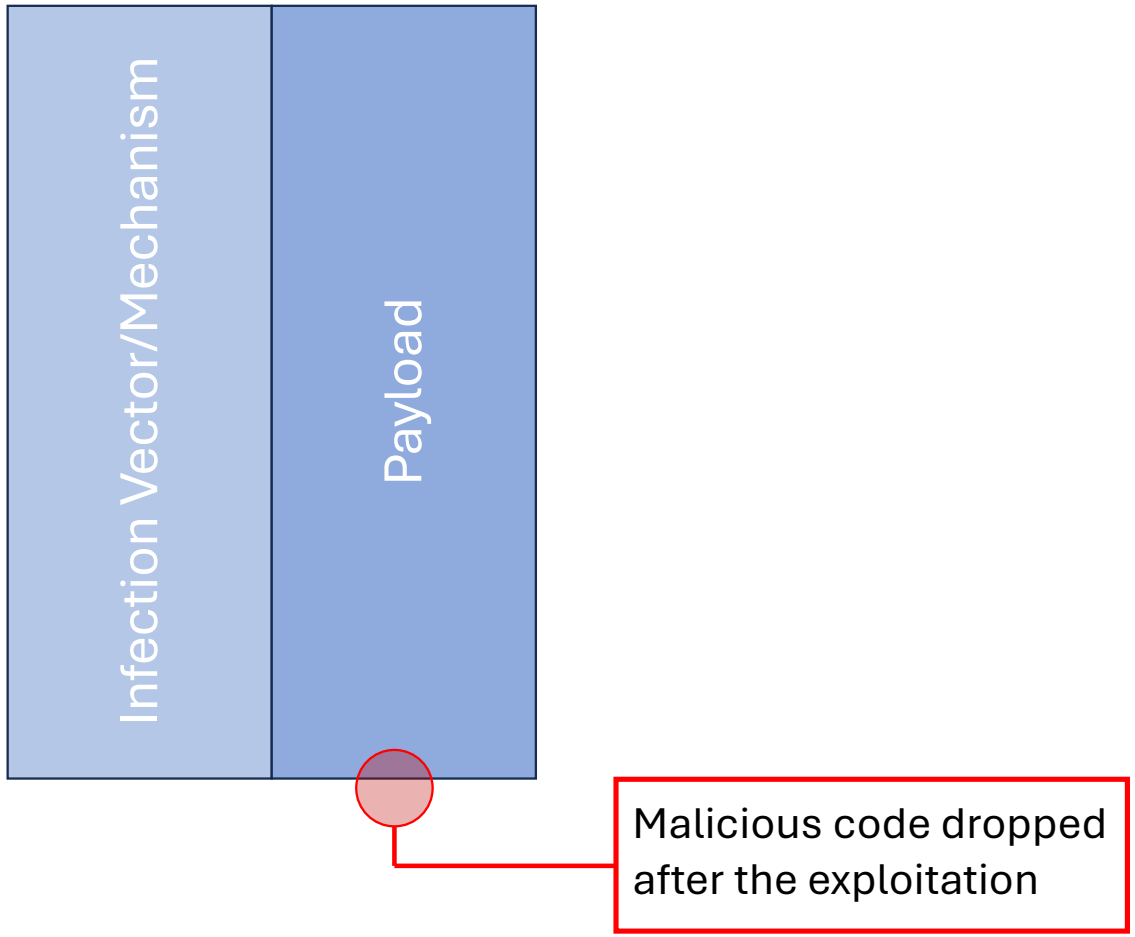


Possible Classification

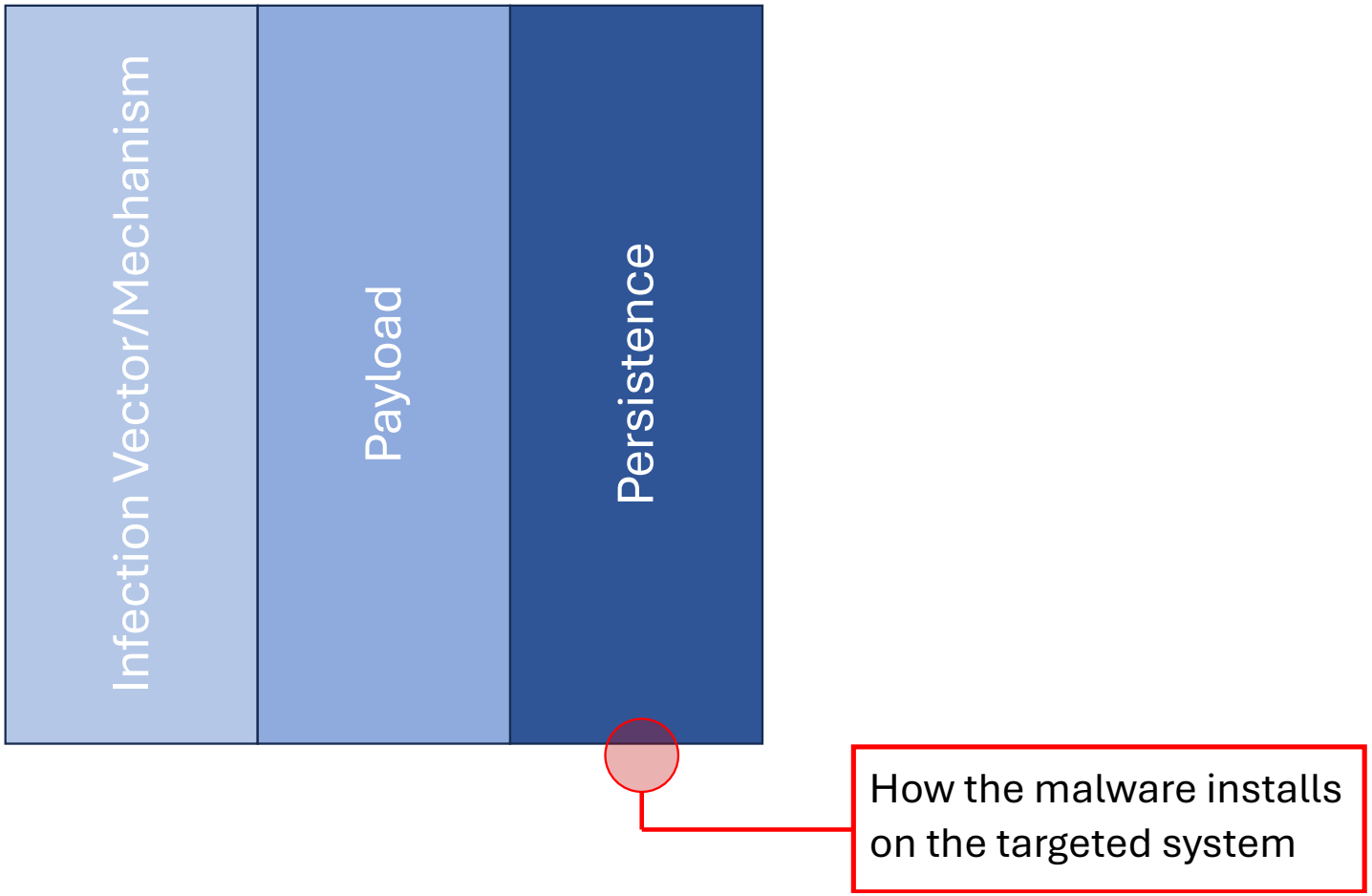
Malware Architecture



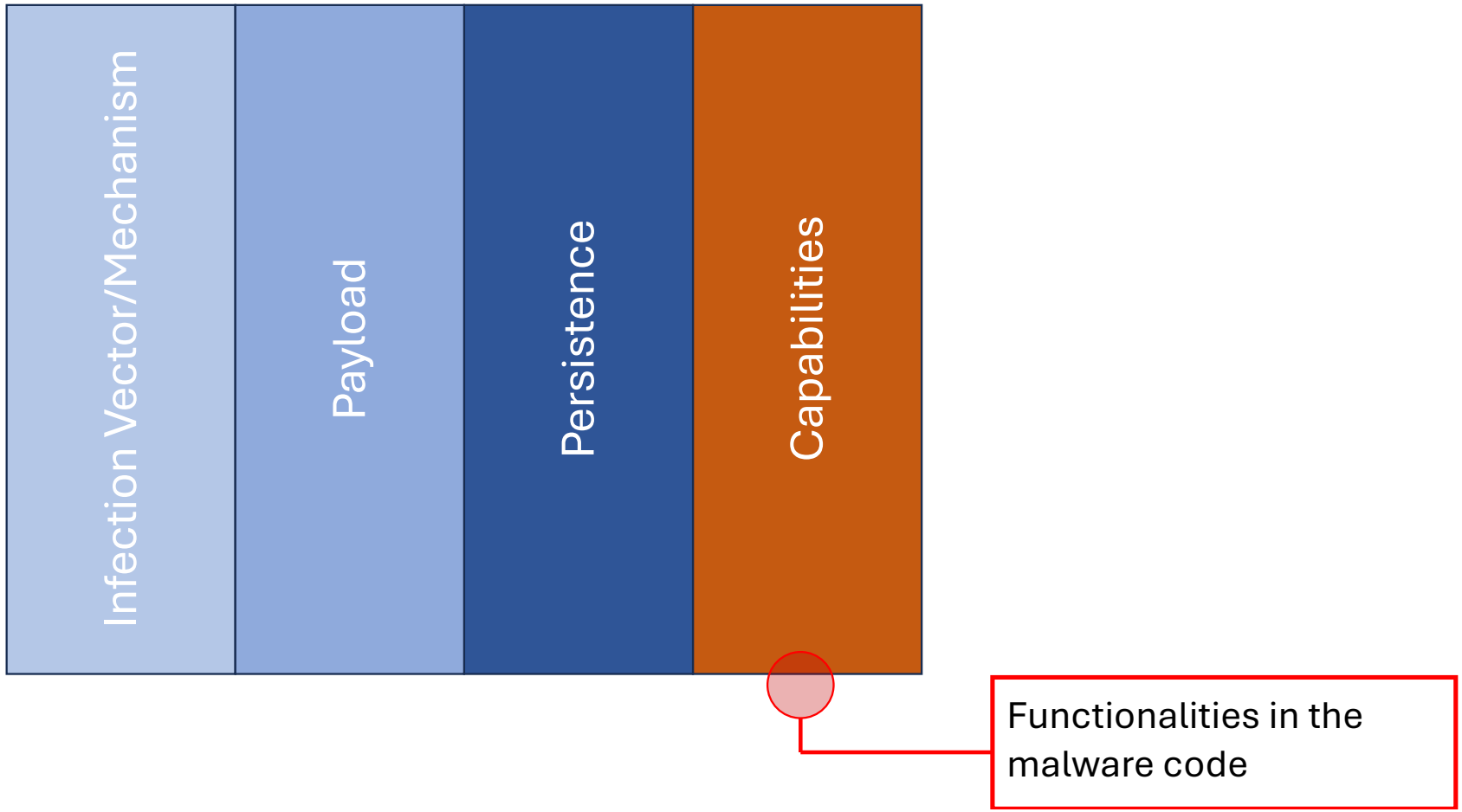
Malware Architecture



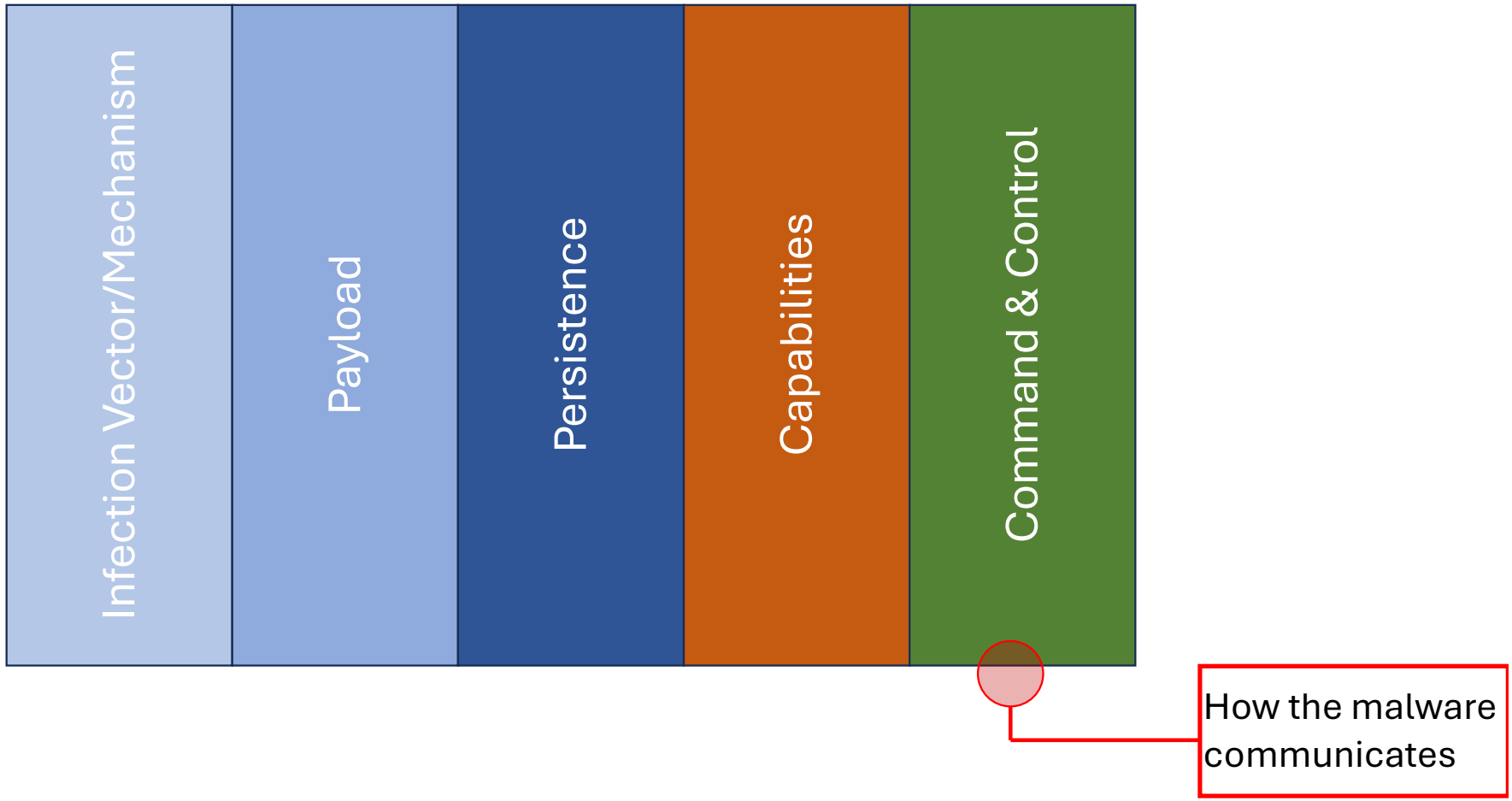
Malware Architecture



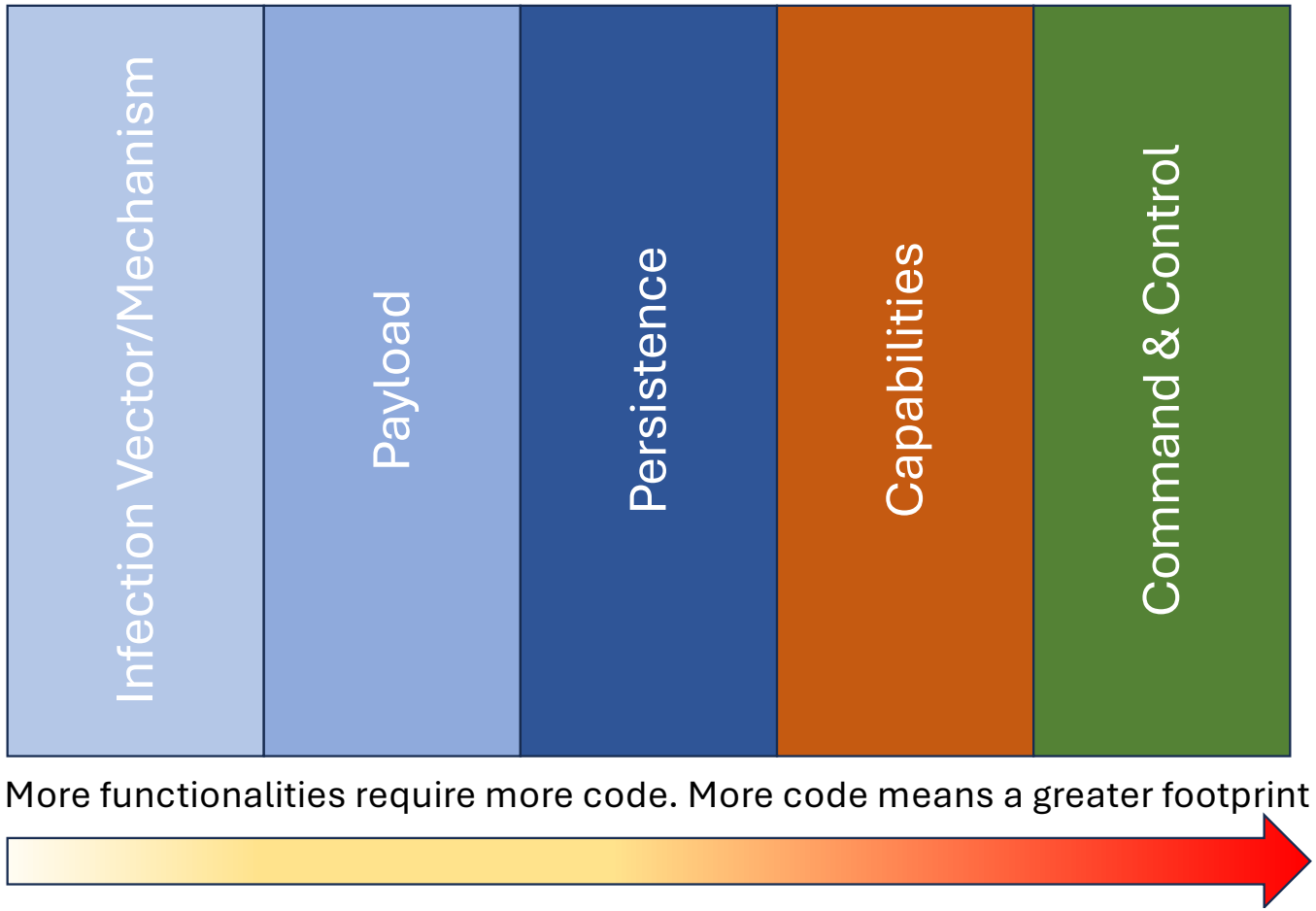
Malware Architecture



Malware Architecture

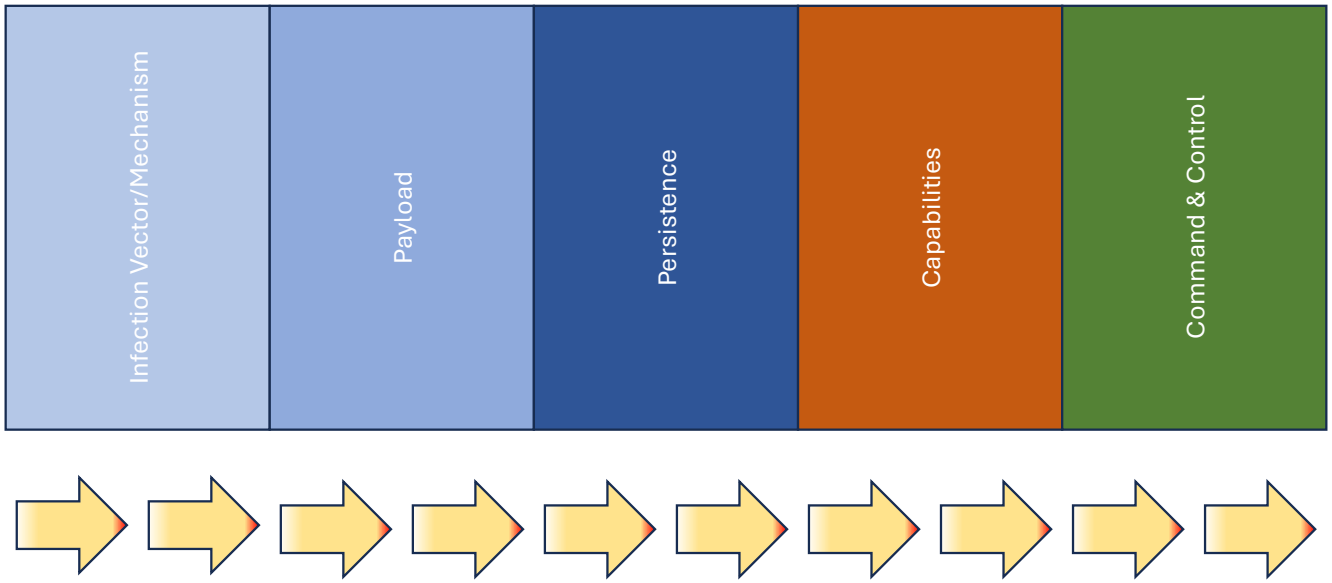


Malware Architecture



Not all malware share the same architecture or implement a rich-set of functionalities

Malware Architecture



Multi-stage mechanisms: the retrieval of a functionalities is delayed until needed, implemented in another artifact, or remotely retrieved. Many most sophisticated threats are not implemented in a monolithic manner.

Examples: win.ssload, GhostWeaver, and StealC

Malware Analysis

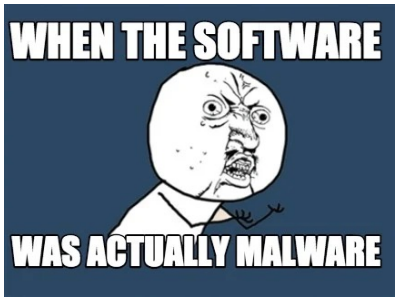
- Malware analysis investigates the behavior of malicious software for:
  - detection
  - mitigation
  - elimination.
- Understanding functionalities of malicious software is important for:
  - tracking its origin, e.g., organized groups or states
  - determining its purposes and aims, e.g., keylogger or RAT
  - outlining motivations and intentions of the attacker, e.g., moneys or geopolitics
  - tracking its host and network activities for distilling signatures
  - designing countermeasures.

Malware Analysis

- Why analyzing a malware? Well, for:
  - incident response
  - develop defensive security approaches
  - forensics and investigative tasks
  - reveal unknown vulnerabilities
  - develop signatures and support detection
  - definition of Indicators of Compromise (IoCs)
  - fun!
- There are **two** main paradigms for **analyzing** a malware:
  - **static analysis**: the binary is analyzed without being executed, mainly for finding code patterns, signatures and IoCs
  - **dynamic analysis**: the binary is executed in a controlled/isolated environment to observe its behaviors.

Malware Analysis

- The main **techniques** for analyzing a malware are:
  - **reverse engineering**: aims at disassembling and/or decompiling the code of a malicious software to grasp its internals and used techniques
  - **network analysis**: focuses on the monitoring of generated network traffic to understand C&C | C2 features, identify remote servers and possible communication patterns
  - **memory forensics**: considers the RAM of the host running the malicious software to evaluate what happens after the infection or to reveal evasive techniques
  - **behavioral analysis**: observes changes in the infected system, for instance interactions with the Windows registry or the creation of new processes/files.



By Aleksa Tamburkovski

(A Quick Journey Into) Static Analysis

- As said, static analysis:
  - inspects a suspect file without its execution
  - collects information useful for its classification
  - provides a prime setting for specific investigation efforts.
- The various steps can be explored via examples.
- Choice:
  - showing main information to be collected
  - the proposed steps are not carved in stone
  - introducing basic tools
  - have some background for further studies.

(A Quick Journey Into) Static Analysis

- Four examples:
  - **Jigsaw**: ransomware created in 2016 and targeting Windows, it has a size of 284 Kbyte and is written in VB.net. Upon activated, it encrypts the MBR of the victim and all the files. Reboots or delays in paying the ransom leads to some files wiped out
  - **Linux.Encoder**: ransomware discovered in 2015, it targets Linux and can be executed via a flaw in the Magento content management framework. Upon activated, it encrypts some files on both local and network drives
  - **Zeus/Zbot**: trojan created in 2007, it targets Windows and steals bank information and logs keystrokes. Usually delivered via drive-by downloads and phishing schemes, here we consider the dropper
  - **SpyEye**: malware appeared in 2016 and targeting Windows, it logs keystroke and grabs information from forms with the aim of stealing credentials.

(A Quick Journey Into) Static Analysis

- The first step involves the **creation of a fingerprint**.
- Fingerprinting is typically done via cryptographic hash values:
  - MD5
  - SHA1
  - SHA256
- Fingerprints can be used to:
  - filenames may change but the hash remains the same
  - if executed (i.e., in dynamic analysis) the malware may change itself
  - de-facto standard identifiers for sharing information among security experts
  - can help to track a sample in databases or antivirus services.

(A Quick Journey Into) Static Analysis

- How to calculate hashes:
  - *md5 target*
  - *sha1 target*
  - *sha256 target*
- Examples:
  - *md5 jigsaw*: 2773e3dc59472296cb0024ba7715a64e
  - *sha1 jigsaw*: 27d99fbca067f478bb91cdbcb92f13a828b00859
  - *sha256 jigsaw*: 3ae96f73d805e1d3995253db4d910300d8442ea603737a1428b613061e7f61e7
- Try to Google them…

(A Quick Journey Into) Static Analysis

- The second step involves the **identification** of the **file type**:
  - filename and extensions should not be considered reliable (e.g., easy to change)
  - the type hints at the target operating system (e.g., Windows or macOS)
  - the architecture hints at the target platform (e.g., 32 bit or 64 bit)
  - discover the real format of executable files in Windows (e.g., .exe, .dll, and .drv)
- How to identify a file type.
- Method 1:
  - via the **file** utility usually shipped in Unix-like systems
  - *man file* for more information.

(A Quick Journey Into) Static Analysis

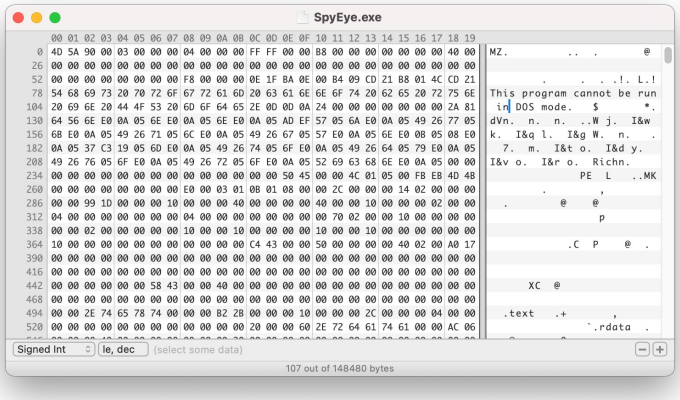
- Method 1:
  - via the **file** utility usually shipped in Unix-like systems
  - *man file* for more information.
- Examples:
  - *file Jigsaw*:
    PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
  - *file Linux.Encoder*:
    ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
  - *file invoice[...].pdf.exe*:
    PE32 executable (GUI) Intel 80386, for MS Windows
  - *file SpyEye.exe*:
    PE32 executable (GUI) Intel 80386, for MS Windows
  - *file SpyEyeUPX*:
    PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed

(A Quick Journey Into) Static Analysis

- Method 2:
  - files can be “manually” inspected and searched for a signature
  - the file signature is a unique sequence of bytes withing the header of the file
  - usually in the first 2-4 bytes of a file.
- Major hex signatures:
  - Windows Portable Executable (PE) files: 4D 5D (or MZ in ASCII)
  - ELF: 7F 45 4C 46
  - Mach-O binary: CF FA ED FE
- To obtain the hex dump of a file:
  - use *xxd*
  - use a hex editor like Hex Fiend (*https://hexfiend.com*).

(A Quick Journey Into) Static Analysis

- Get file signature with *xxd*:
  - *xxd -g 1 -l 16 target* (g defines how separate bytes, l the number of line to show).
- Examples:
  - *xxd -g 1 -l 16 Jigsaw:*
    4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
  - xxd -g 1 -l 16 Linux.Encoder:
    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

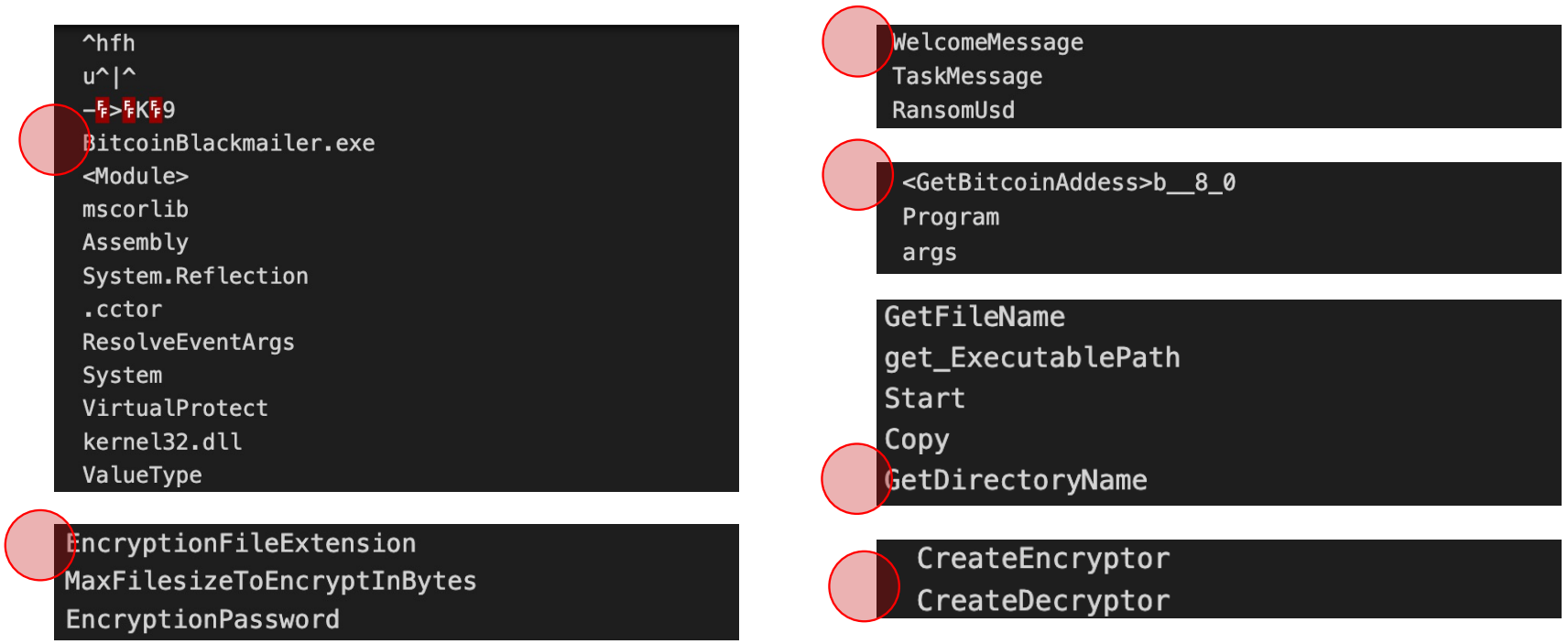


(A Quick Journey Into) Static Analysis

- The third step aims at **extracting** ASCII **strings**:
  - strings may leak information on internals (e.g., contacted IP addresses)
  - textual messages (e.g., taunts!)
  - comments and debug messages left by the malware developers.
- An efficient method is to use:
  - the string utility usually shipped in Unix-like systems
  - *man strings* for more information.

(A Quick Journey Into) Static Analysis

- Example:
  - strings Jigsaw



(A Quick Journey Into) Static Analysis

- Example:
  - strings Linux.Encoder

```
 ! mbedtls_pk_decrypt returned -0x%04x
failed
 ! mbedtls_pk_encrypt returned -0x%04x
./index.crypto
./readme.crypto
Decrypting file: %s
%s/%s
.encrypted
failed
 ! mbedtls_ctr_drbg_seed returned -0x%04x
error loading %s
/README_FOR_DECRYPT.txt
Start encrypting...
decrypt
Start decrypting...
mbedtls_pk_encrypt
/root/.ssh
/usr/bin
/etc/ssh
/home
/root
/var/lib/mysql
/var/www
```

```
-----END RSA PRIVATE KEY-----
-----END EC PRIVATE KEY-----
-----END PRIVATE KEY-----
-----BEGIN PRIVATE KEY-----
-----END PUBLIC KEY-----
-----BEGIN PUBLIC KEY-----
-----BEGIN
-----BEGIN RSA PRIVATE KEY-----
-----BEGIN EC PRIVATE KEY-----
-----END ENCRYPTED PRIVATE KEY-----
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

```
.html
.tar
.sql
.css
.pdf
.tgz
.war
.jar
.java
.class
```

(A Quick Journey Into) Static Analysis

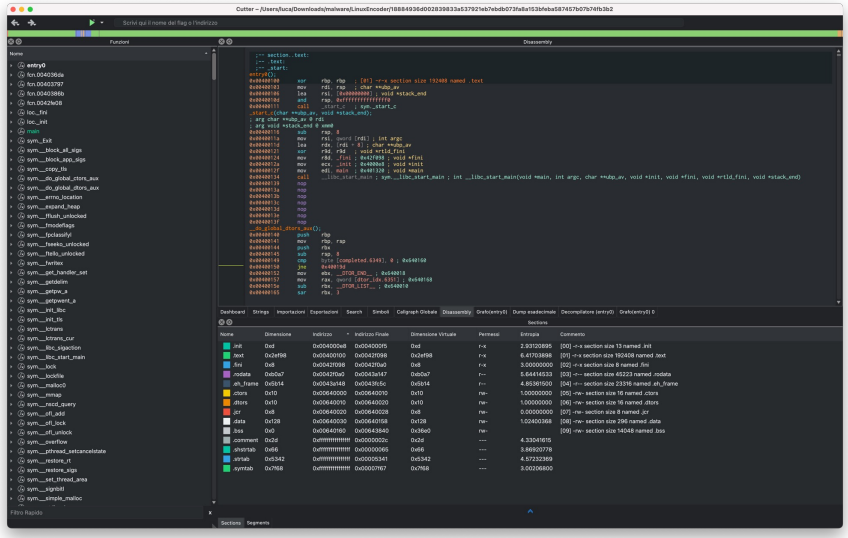
- Example:
  - strings Zeus

```
GetClipboardOwner
CallWindowProcW
GetProcessDefaultLayout
UpdateWindow
AllowSetForegroundWindow
AppendMenuA
GetCaretPos
GetSysColor
DestroyCursor
GetClipboardData
```

```
SHLWAPI.PathMakeSystemFolderW
BemaCadsPodsWavyCedeRadsbrioOustPerefenom
USER32.SetDlgItemTextW
BullbonyaweeWaitsnugTierDriblibye
KERNEL32.VirtualQuery
CameValeWauler
USER32.IsIconic
CedeSalsshulLimyThroliraValeDonabox
USER32.CreateCaret
CellrotoCrudUntohighCols
KERNEL32.CreateFileA
DenyLubeDunssawsOresvarut
SHLWAPI.PathRemoveFileSpecA
DragRoutflusCrowPeatmownNewsyaksSerfmare
USER32.DestroyIcon
Dumpcotsavo
USER32.SetDlgItemInt
DungBadebankBangGelthoboCocaBozotsksWheyVaryShoghoseNipsCadisi
USER32.EndPaint
ExitRollWoodGumsgamaSloerevsWussletssinkYearZitiryesHypout
USER32.GetClassInfoW
FociTalcileador
KERNEL32.ConvertDefaultLocale
```

Static Analysis: Cutter

- Cutter is a free and open-source reverse engineering framework:
  - based on Rizin, *https://rizin.re*
  - many functionalities wrapped in a unique place
  - *https://cutter.re*
  - on macOS: *brew install --cask cutter*
  - give it a try!
- If you are hardcore, you can also try:
  - radare2
  - *https://www.radare.org/n/*
  - ...be patient :)



(A Quick Journey Into) Static Analysis

- The third step considers the analysis of **dependencies** and **imports**:
  - this reveals the used Dynamic Link Libraries (DLLs on Windows)
  - gives a general idea on functionalities deployed by the malware
  - hints at capabilities and offensive behaviors
  - several tools can do this
  - in the following, screenshots are taken from Cutter.

(A Quick Journey Into) Static Analysis

- Example:
  - SpyEye

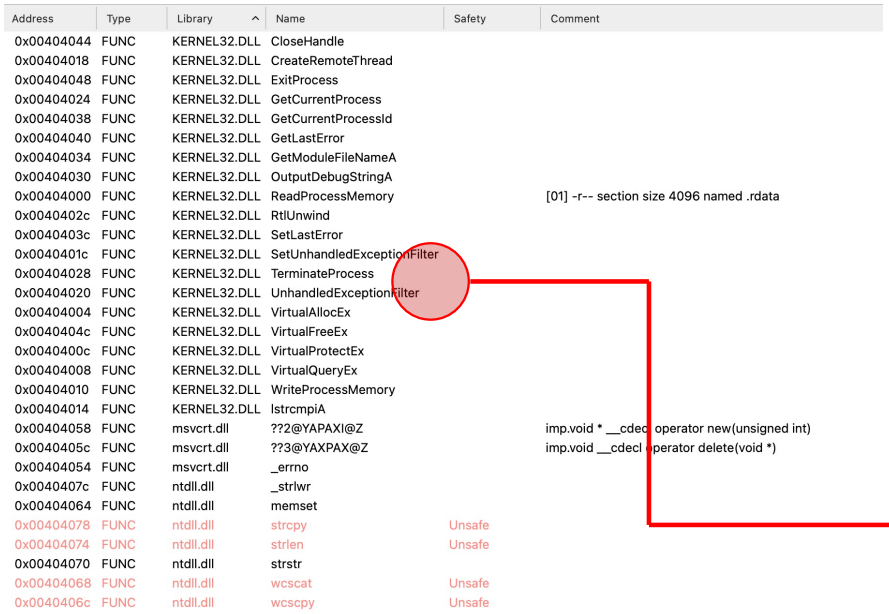
Address	Type	Library	Name	Safety	Comment
0x00404044	FUNC	KERNEL32.DLL	CloseHandle		
0x00404018	FUNC	KERNEL32.DLL	CreateRemoteThread		
0x00404048	FUNC	KERNEL32.DLL	ExitProcess		
0x00404024	FUNC	KERNEL32.DLL	GetCurrentProcess		
0x00404038	FUNC	KERNEL32.DLL	GetCurrentProcessId		
0x00404040	FUNC	KERNEL32.DLL	GetLastError		
0x00404034	FUNC	KERNEL32.DLL	GetModuleFileNameA		
0x00404030	FUNC	KERNEL32.DLL	OutputDebugStringA		
0x00404000	FUNC	KERNEL32.DLL	ReadProcessMemory		[01] -r-- section size 4096 named .rdata
0x0040402c	FUNC	KERNEL32.DLL	RtlUnwind		
0x0040403c	FUNC	KERNEL32.DLL	SetLastError		
0x0040401c	FUNC	KERNEL32.DLL	SetUnhandledExceptionFilter		
0x00404028	FUNC	KERNEL32.DLL	TerminateProcess		
0x00404020	FUNC	KERNEL32.DLL	UnhandledExceptionFilter		
0x00404004	FUNC	KERNEL32.DLL	VirtualAllocEx		
0x0040404c	FUNC	KERNEL32.DLL	VirtualFreeEx		
0x0040400c	FUNC	KERNEL32.DLL	VirtualProtectEx		
0x00404008	FUNC	KERNEL32.DLL	VirtualQueryEx		
0x00404010	FUNC	KERNEL32.DLL	WriteProcessMemory		
0x00404014	FUNC	KERNEL32.DLL	lstrcmpiA		
0x00404058	FUNC	msvcrt.dll	??2@YAPAXI@Z		imp.void * __cdecl operator new(unsigned int)
0x0040405c	FUNC	msvcrt.dll	??3@YAXPAX@Z		imp.void __cdecl operator delete(void *)
0x00404054	FUNC	msvcrt.dll	_errno		
0x0040407c	FUNC	ntdll.dll	_strlwr		
0x00404064	FUNC	ntdll.dll	memset		
0x00404078	FUNC	ntdll.dll	strcpy	Unsafe	
0x00404074	FUNC	ntdll.dll	strlen	Unsafe	
0x00404070	FUNC	ntdll.dll	strstr		
0x00404068	FUNC	ntdll.dll	wcscat	Unsafe	
0x0040406c	FUNC	ntdll.dll	wcscpy	Unsafe	

Kernel32.dll provides core functionalities, such as memory management, threads handling, I/O, and mechanisms to retrieve system information. Is a legitimate library but offers the malware several capabilities.

msvcrt.dll contains the C runtime and includes string manipulations and memory allocation.

(A Quick Journey Into) Static Analysis

- Example:
  - SpyEye and SpyEye packed with UPX.



Address	Type	Library	Name	Safety	Comment
0x00404044	FUNC	KERNEL32.DLL	CloseHandle		
0x00404018	FUNC	KERNEL32.DLL	CreateRemoteThread		
0x00404048	FUNC	KERNEL32.DLL	ExitProcess		
0x00404024	FUNC	KERNEL32.DLL	GetCurrentProcess		
0x00404038	FUNC	KERNEL32.DLL	GetCurrentProcessId		
0x00404040	FUNC	KERNEL32.DLL	GetLastError		
0x00404034	FUNC	KERNEL32.DLL	GetModuleFileNameA		
0x00404030	FUNC	KERNEL32.DLL	OutputDebugStringA		
0x00404000	FUNC	KERNEL32.DLL	ReadProcessMemory		[01] -r-- section size 4096 named .rdata
0x0040402c	FUNC	KERNEL32.DLL	RtlUnwind		
0x0040403c	FUNC	KERNEL32.DLL	SetLastError		
0x0040401c	FUNC	KERNEL32.DLL	SetUnhandledExceptionFilter		
0x00404028	FUNC	KERNEL32.DLL	TerminateProcess		
0x00404020	FUNC	KERNEL32.DLL	UnhandledExceptionFilter		
0x00404004	FUNC	KERNEL32.DLL	VirtualAllocEx		
0x0040404c	FUNC	KERNEL32.DLL	VirtualFreeEx		
0x0040400c	FUNC	KERNEL32.DLL	VirtualProtectEx		
0x00404008	FUNC	KERNEL32.DLL	VirtualQueryEx		
0x00404010	FUNC	KERNEL32.DLL	WriteProcessMemory		
0x00404014	FUNC	KERNEL32.DLL	lstrcmpiA		
0x00404058	FUNC	msvcrt.dll	??2@YAPAXI@Z		imp.void * __cdecl operator new(unsigned int)
0x0040405c	FUNC	msvcrt.dll	??3@YAXPAX@Z		imp.void __cdecl operator delete(void *)
0x00404054	FUNC	msvcrt.dll	_errno		
0x0040407c	FUNC	ntdll.dll	_strlwr		
0x00404064	FUNC	ntdll.dll	memset		
0x00404078	FUNC	ntdll.dll	strcpy	Unsafe	
0x00404074	FUNC	ntdll.dll	strlen	Unsafe	
0x00404070	FUNC	ntdll.dll	strstr		
0x00404068	FUNC	ntdll.dll	wcscat	Unsafe	
0x0040406c	FUNC	ntdll.dll	wcscpy	Unsafe	

Unpacked

Address	Type	Library	Name	Safety	Comment
0x0042a814	FUNC	KERNEL32.DLL	ExitProcess		
0x0042a804	FUNC	KERNEL32.DLL	GetProcAddress		
0x0042a800	FUNC	KERNEL32.DLL	LoadLibraryA		
0x0042a80c	FUNC	KERNEL32.DLL	VirtualAlloc		
0x0042a810	FUNC	KERNEL32.DLL	VirtualFree		
0x0042a808	FUNC	KERNEL32.DLL	VirtualProtect		
0x0042a81c	FUNC	msvcrt.dll	_errno		
0x0042a824	FUNC	ntdll.dll	memset		

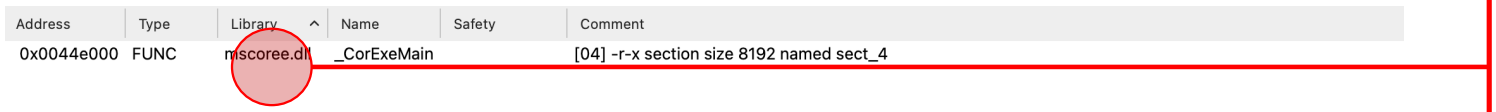
Packed

Malware can also load a DLL at **runtime** via **LoadLibrary()** and can resolve the function address via **GetProcessAdress()**. Since loaded at runtime, the used functionalities will not be present in the import table of the PE file.

Hint (not always true): very few imports often suggest that the malware has been obfuscated.

(A Quick Journey Into) Static Analysis

- Example:
  - JigSaw

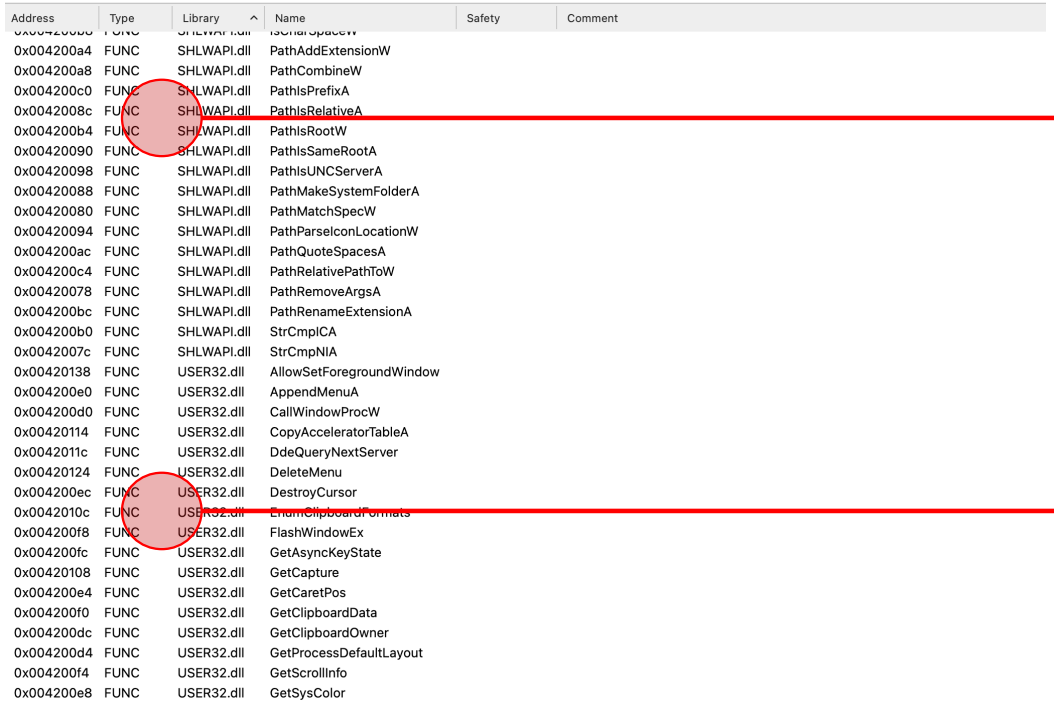


Address	Type	Library	Name	Safety	Comment
0x0044e000	FUNC	mscoree.dll	_CorExeMain		[04] -r-x section size 8192 named sect_4

mscoree.dll contains core functionalities of the .NET framework.

(A Quick Journey Into) Static Analysis

- Example:
  - Zeus



Address	Type	Library	Name	Safety	Comment
0x004200a4	FUNC	SHLWAPI.dll	PathAddExtensionW		
0x004200a8	FUNC	SHLWAPI.dll	PathCombineW		
0x004200c0	FUNC	SHLWAPI.dll	PathIsPrefixA		
0x0042008c	FUNC	SHLWAPI.dll	PathIsRelativeA		
0x004200b4	FUNC	SHLWAPI.dll	PathIsRootW		
0x00420090	FUNC	SHLWAPI.dll	PathIsSameRootA		
0x00420098	FUNC	SHLWAPI.dll	PathIsUNCServerA		
0x00420088	FUNC	SHLWAPI.dll	PathMakeSystemFolderA		
0x00420080	FUNC	SHLWAPI.dll	PathMatchSpecW		
0x00420094	FUNC	SHLWAPI.dll	PathParseIconLocationW		
0x004200ac	FUNC	SHLWAPI.dll	PathQuoteSpacesA		
0x004200c4	FUNC	SHLWAPI.dll	PathRelativePathToW		
0x00420078	FUNC	SHLWAPI.dll	PathRemoveArgsA		
0x004200bc	FUNC	SHLWAPI.dll	PathRenameExtensionA		
0x004200b0	FUNC	SHLWAPI.dll	StrCmpICA		
0x0042007c	FUNC	SHLWAPI.dll	StrCmpNIA		
0x00420138	FUNC	USER32.dll	AllowSetForegroundWindow		
0x004200e0	FUNC	USER32.dll	AppendMenuA		
0x004200d0	FUNC	USER32.dll	CallWindowProcW		
0x00420114	FUNC	USER32.dll	CopyAcceleratorTableA		
0x0042011c	FUNC	USER32.dll	DdeQueryNextServer		
0x00420124	FUNC	USER32.dll	DeleteMenu		
0x004200ec	FUNC	USER32.dll	DestroyCursor		
0x0042010c	FUNC	USER32.dll	EnumClipboardFormats		
0x004200f8	FUNC	USER32.dll	FlashWindowEx		
0x004200fc	FUNC	USER32.dll	GetAsyncKeyState		
0x00420108	FUNC	USER32.dll	GetCapture		
0x004200e4	FUNC	USER32.dll	GetCaretPos		
0x004200f0	FUNC	USER32.dll	GetClipboardData		
0x004200dc	FUNC	USER32.dll	GetClipboardOwner		
0x004200d4	FUNC	USER32.dll	GetProcessDefaultLayout		
0x004200f4	FUNC	USER32.dll	GetScrollInfo		
0x004200e8	FUNC	USER32.dll	GetSysColor		

SHLWAPI.dll offers several utility functions. In this case, malware uses path manipulation capabilities and possible check on files/paths present on the host of the victim.

USER32.dll delivers core functions for managing the GUI and interacting with the OS. Used functions suggest the “spyware” nature of Zeus, for instance by acting on the clipboard or on states of windows.

(A Quick Journey Into) Static Analysis

- The fourth step can be an attempt of understanding the code:
  - reverse engineering is not a simple thing!
  - we discuss it here just to have a pointer for the future
  - used as an excuse to look at Radare2 and Cutter again.
- Radare2 super-mega minimal how-to:
  - *radare2 target*
  - ***aaa*** to analyze the program
  - ***afl*** to list all the functions
  - ***s name*** to seek to the function name
  - ***pdf; agf*** disassembly and print an ASCII graph
  - ***q*** to quit
  - the learning curve is a bit challenging…

(A Quick Journey Into) Static Analysis

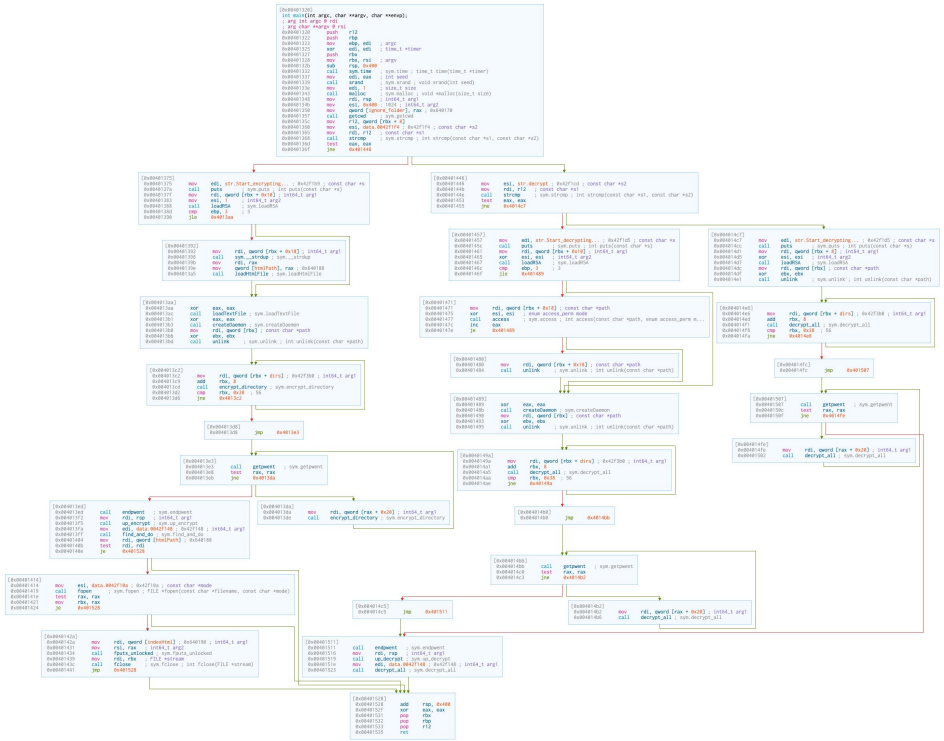
- Example Linux.Encoder

Branch for encrypting files

Branch for decrypting files

(A Quick Journey Into) Static Analysis

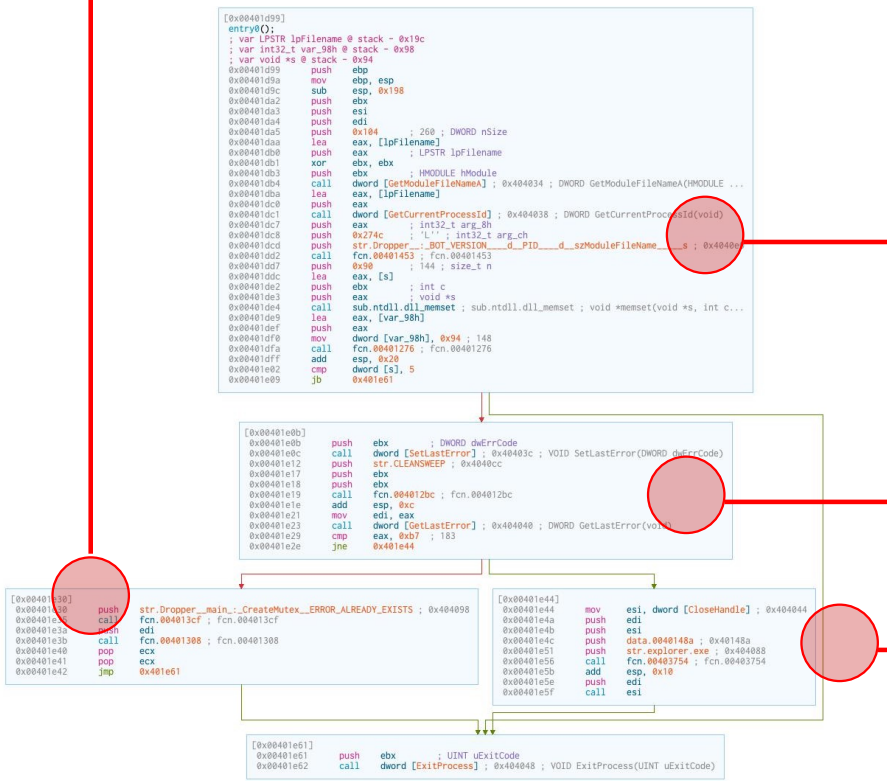
- Example Linux.Encoder



main exported via Cutter

(A Quick Journey Into) Static Analysis

- Example SpyEye



If another instance of the malware is found, the malware aborts to avoid over-proliferation

The malware identifies itself also to transmit information to the C2 server

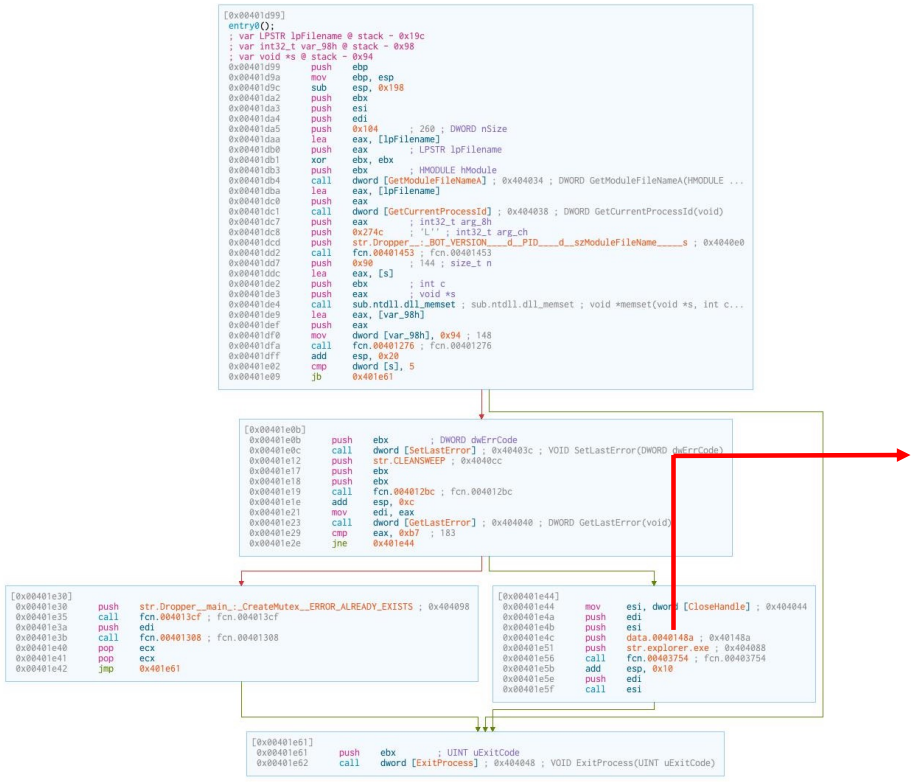
It creates a mutex to signal its presence and that it is running

If it decides to run, it invokes Internet Explorer

entry0 exported via Cutter

(A Quick Journey Into) Static Analysis

- Example SpyEye

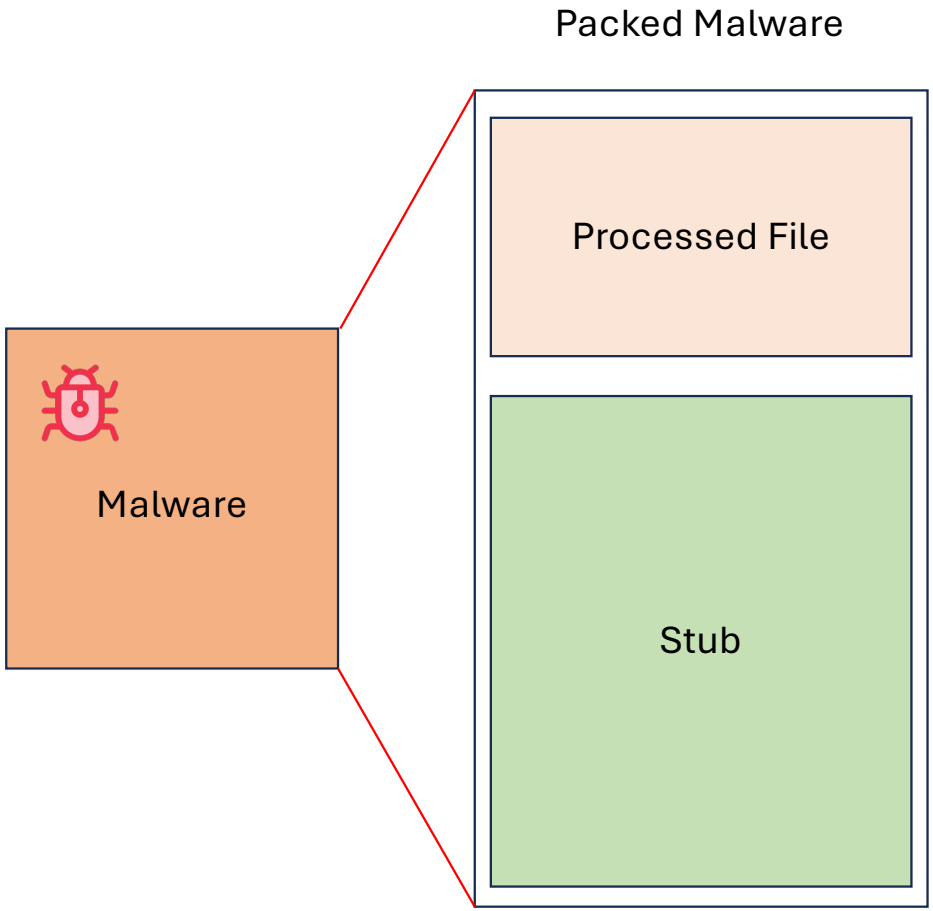


Commands contained in data.0040148a and passed to explorer.exe

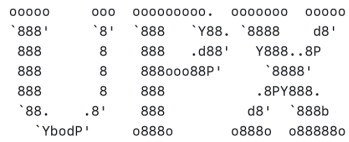
entry0 exported via Cutter

Packers

- **Packers** allow to **compress** a file and (also) **disguise** its content.
- Roughly, there are **two** main types of packers:
  - **archive**: files are compressed in an archive (e.g., zip or rar), which may only contain the payload or also an installer (e.g., phishing and Lu0bot)
  - **UPX**: files are compressed and obfuscated via encryption (e.g., GhOstRAT and AgentTesla).



Packers: UPX



- The Ultimate Packer for eXecutables (UPX):
  - first introduced in 1996 by Oberhumer, Molnar, and Reiser
  - available at: *https://github.com/upx/upx*
  - for more information: *man upx*
  - **example**: *upx test -o testpacked*
  - warning: macOS yet not supported, will require *--force-macos*
- The presence of UPX can be spotted via:
  - analyzing strings: *strings target* | *grep upx* or *strings target* | *grep UPX!*
  - by inspecting the hexdump: *hexdump –C target* | *grep UPX*

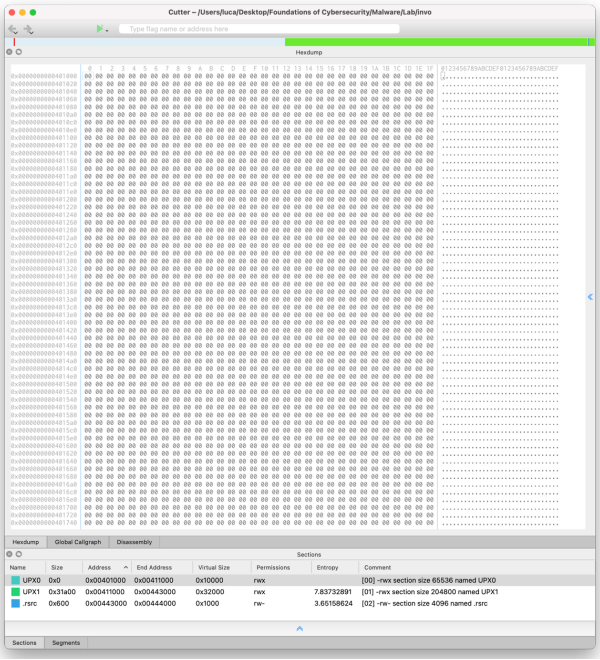
```
➜  Lab hexdump -C testpacked | grep UPX
00000410  00 00 00 00 00 00 00 00  77 f3 d8 7e 55 50 58 21  |........w..~UPX!|
00000810  00 80 00 40 02 00 ff 00  00 00 00 55 50 58 21 00  |...@.......UPX!.|
000009f0  68 65 20 55 50 58 20 54  65 61 6d 2e 20 41 6c 6c  |he UPX Team. All|
000013e0  41 09 00 00 00 ff 55 50  58 21 0d 25 02 08 be 3f  |A.....UPX!.%...?|
```

ASCII conversion

```
ooooo     ooo  ooooooooo.   ooooooo  ooooo
`888'     `8'  `888   `Y88. `8888    d8'
 888       8    888   .d88'  Y888..8P
 888       8    888ooo88P'     `8888'
 888       8    888           .8PY888.
 `88.     .8'   888          d8'  `888b
  `YbodP'      o888o        o888o  o88888o
```

Packers: UPX

- The presence of UPX can be spotted via:
  - an **empty section** named **UPX0** without data and a large virtual memory entry
  - a **stub section** named **UPX1** containing the compressed executable.



Try also: *hexdump -C target*

Dynamic Analysis

- Dynamic analysis of malware consists in running a sample in a controlled and isolated environment for:
  - monitoring activities
  - interactions
  - changes in the observed system.
- When executing a malware, there are **four** major types of monitoring:
  - **process**: examines the properties and activities of processes
  - **file system**: examines the activities over the file system
  - **network**: examines traffic from/to the system
  - **registry (Windows)**: examines the keys read/modified.

Dynamic Analysis

- Not enough time for discussing dynamic analysis in this course :(
- A tool to know is the **Cuckoo Sandbox**:
  - started at the Google Summer of Code in 2010
  - available at *https://github.com/cuckoosandbox*
  - automatically run samples and provides various analyses.
- Information about the malware collected by Cuckoo:
  - calls performed by the spawned processes
  - files created and downloaded
  - (full and partial) memory dumps
  - traffic traces in .pcap format
  - screenshots.

Anti-forensics

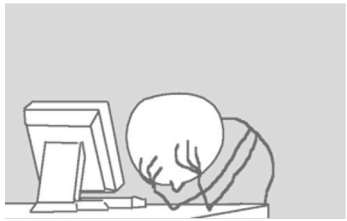
- Recap: malicious software can be:
  - disassembled and reversed
  - observed at runtime.
- Malware (and malware developers) do not just sit around doing nothing!
- **Anti-forensics** is an umbrella for techniques used by malware to.
  - impair or beat forensics investigations and analysis
  - conceal malicious activities
  - prevent detection and trace back the attacker.

Anti-forensics

- Main techniques that malware can deploy to resist against investigations are:
  - **timestomping**: time and date when a file is created | accessed | modified | executed are changed
  - **encryption**: configuration files | payloads | network flows | are encrypted to evade detection
  - **anti-debug**: mechanisms to make debug harder, e.g., hiding in the ToP or removing symbols
  - **wiping**: after performed the attack (e.g., stealing data), malware artifacts are promptly deleted
  - **trail obfuscation**: decoy temporary data within a burden of irrelevant files or folders
  - **anti-sandbox**: when a sandbox is detected, the malware behaves differently, e.g., it does not deploy any attack routine
  - **anti-virtualization**: similar to anti-sandbox but against virtual machines
  - **information hiding**: more in Module 6.

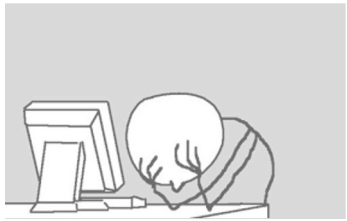
Anti-forensics: Anti-Sandbox/Anti-Virtualization

- Let us consider anti-sandbox/VM as an example of anti-forensics.
- How can a malware understand that is:
  - **sandboxed** (probably for **dynamic analysis** purposes)?
- Sometimes it is easier than you think.



Anti-forensics: Anti-Sandbox/Anti-Virtualization

- Let us consider anti-sandbox/VM as an example of anti-forensics.
- How can a malware understand that is:
  - **sandboxed** (probably for **dynamic analysis** purposes)?
- Sometimes it is easier than you think.
- Malware can "reason" by simply collecting:
  - **general information**: username, hostname and storage units often use identifiers like, *cuckoo*, *test*, *QEMU*, *VMWare*, or *virtual HD*
  - **basic hardware stats**: lack of audio devices, well-known prefixes in MAC address (e.g., *00:1C:42* for *Parallels*, *08:00:27* for *Virtual Box*)
  - **runtime information**: presence of directories like *virtualbox guest additions*\ or processes like *vmsrvc.exe*
  - **hardware limitations**: failure to get measurements from temperature sensors or "bogus" CPUID (e.g., *XenVMMXenVMM* for *Xen*).



YARA Rules

- Yet Another Recursive Acronym (YARA) is a tool:
  - for helping researchers to **identify** and **classify** malware
  - based on textual descriptions
  - introduced by VirusTotal
  - available at: *https://virustotal.github.io/yara/*
  - widely-used, simple and powerful.
- How YARA works:
  - **basic idea**: a malware sample may contain several **unique** indicators (e.g., strings or binary patterns)
  - indicators are “described” via a **textual rule**
  - an engine **applies the rules** over the target binary(ies).

YARA Rules

- There are two versions of YARA!
- YARA:
  - currently in “maintenance mode”, i.e., only bug fixes and no new features
  - available at: *https://github.com/VirusTotal/yara*
  - it is strongly encouraged to migrate to the newer version.
- YARA-X:
  - a complete rework/rewrite of YARA in Rust
  - considered stable from January 2025 (YARA-X 1.0.0)
  - available at: *https://github.com/VirusTotal/yara-x*
  - if you are new to YARA, you should start here.

YARA-X

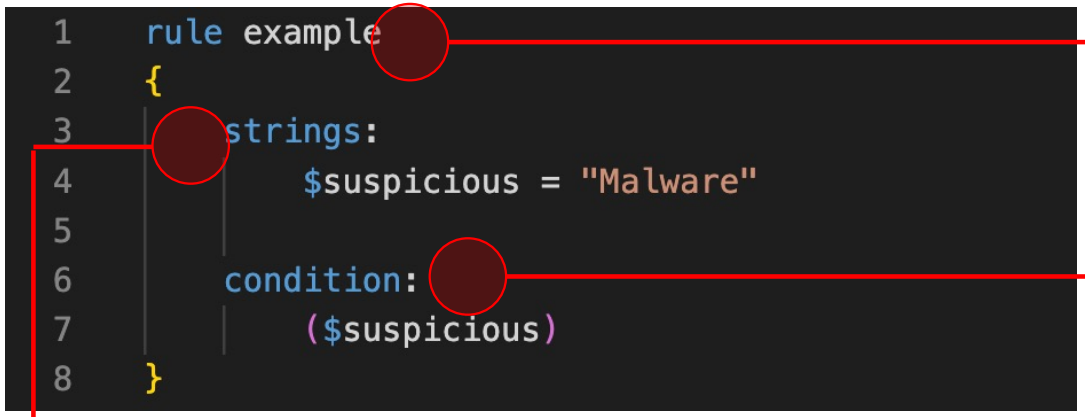
YARA Rules

- YARA vs YARA-X:
  - YARA-X is backward compatible to YARA
  - better error reporting and nicer CLI
  - various performance improvements
  - able to dissect many file formats (e.g., PE, ELF and Mach-O)
  - many APIs are not compatible, programs invoking YARA may need a rewrite
  - for this course: YARA and YARA-X can be used interchangeably.
- Official documentation:
  - *https://yara.readthedocs.io/en/latest/*

YARA Rules

- Installing YARA:
  - macOS: *brew install yara* OR *brew install yara-x*
  - clone the repo and following instructions.
- Launching YARA:
  - *yara –r RULES_PATH TARGET_PATH*
  - *yr scan RULES_PATH TARGET_PATH*
  - if lost: *yara --help* OR *yr --help*
  - have fun!

YARA Rules: Anatomy



```
1   rule example
2   {
3       strings:
4           $suspicious = "Malware"
5
6       condition:
7           ($suspicious)
8   }
```

Rule Identifier: is a case-sensitive entry describing the rule and can contain alphanumeric characters and the underscore. It cannot exceed 128 characters and cannot start with digit.

String Definition: is a section containing strings (i.e., text, hexadecimal, and regular expressions) used within the rule. If the rules is not based on strings, it can be omitted. Each string has an identifier, denoted by **$** and alphanumerical characters (e.g., a sort of variable).

Condition: it is mandatory and it defines the logic of the rule. Boolean expressions can be used to decide whether a rule does/does not match, e.g., **($a or $b)**.

YARA Rules: Anatomy

```
1   rule example_bin
2   {
3       meta:
4       description = "Seek for a malicious string only in Mach-O Binaries"
5
6       strings:
7           $sign = {cf fa ed fe}
8           $suspicious = "Malware"
9
10      condition:
11          ($sign at 0) and $suspicious
12  }
```

Rule Meta Data: it contains additional information about the rule. Optional, but provides valuable information on the rule.

The previous rule seeks information in all files. If we want to limit the search to executable file, we can specify the file signature to inspect.

The utility "*xxd -g 1 -l 4*" can be used to extract file signatures of Mach-O binaries, i.e., CF FA ED FE.

The **at** clause specifies the offset at which the string must be present. The beginning of the file is where signatures are usually placed.

YARA Rules: Anatomy

```
rule example_bin_packed
{
    meta:
    description = "Seek for a malicious string only in UPX-packed Mach-O Binaries"
    reference = "https://upx.github.io"

    strings:
        $sign = {cf fa ed fe}
        $suspicious = "Malware"
        $upx = "UPX!"


    condition:
        ($sign at 0) and $suspicious and $upx
}
```

Refined the metadata also with a reference to UPX to make the rule more informative.

As a signature for UPX, we use the fact that binaries packed with UPX contains the “UPX!” string (see, the output of “*strings* target_file”). However, the signature is very weak, as the rule will fire even in the presence of non-packed binaries.

Warning: as version 5.0.1, UPX does not support Mach-O binaries. They are compressed but cannot be executed. Provided here just for the sake of the example!

YARA Rules: Modules

- Features supported by YARA rules can be extended via modules:
  - allows to create functions and more structured rules
  - can capture more complex conditions.
- Supported modules:
  - **PE**: creation of more fine-grained rules for the **P**ortable **E**xecutable format (Windows)
  - **ELF**: creation of more fine-grained rules for **E**xecutable and **L**inking **F**ormat (Unix)
  - **Cuckoo**: creation of rules based on the Cuckoo sandbox
  - **Magic**: identify the type of file via the output of the "*file*" command
  - **Hash**: for calculating and creating signatures with MD5, SHA1, and SHA256 hashes
  - **Math**: for calculating numerical values from portion of files
  - **Dotnet**: creation of more fine-grained rules for the .NET files
  - **Time**: for setting temporal conditions in a rule
  - **Console**: for logging information during the condition execution
  - **String**: allows to manipulate strings returned by modules
  - **LNK**: creation of more fine-grained rules for LNK files (Windows Shortcut Files)

YARA Rules: Anatomy

```
1   import "hash"
2
3
4   rule example_bin_hash
5   {
6       meta:
7       description = "Seek for a malicious file in Mach-O Binaries via its MD5 signature"
8       author = "Foundations of Cybersecurity"
9       date = "2025-06-24"
10      reference = "https://yara.readthedocs.io/en/stable/modules/hash.html"
11
12      strings:
13          $sign = {cf fa ed fe}
14
15
16      condition:
17          ($sign at 0) and
18          (filesize < 1MB) and
19          hash.md5(0, filesize) == "5689d720e2fc83b8a519970a094e19b4"
20
21  }
```

Import functionalities of the “hash” module

Further refined the metadata section

Limit the analysis to Mach-O files

Search for a specific sample, e.g., the given incarnation of a malware has been isolated and its MD5 computed via “*md5*”.

Consider only files < 1 Mbyte

YARA Rules: SpyEye

```
rule spyeye : banker
{
    meta:
        author = "Jean-Philippe Teissier / @Jipe_"
        description = "SpyEye X.Y memory"
        date = "2012-05-23"
        version = "1.0"
        filetype = "memory"

    strings:
        $spyeye = "SpyEye"
        $a = "%BOTNAME%"
        $b = "globplugins"
        $c = "data_inject"
        $d = "data_before"
        $e = "data_after"
        $f = "data_end"
        $g = "bot_version"
        $h = "bot_guid"
        $i = "TakeBotGuid"
        $j = "TakeGateToCollector"
        $k = "[ERROR] : Omfg! Process is still active? Lets kill that mazafaka!"
        $l = "[ERROR] : Update is not successfull for some reason"
        $m = "[ERROR] : dwErr == %u"
        $n = "GRABBED DATA"

    condition:
        $spyeye or (any of ($a,$b,$c,$d,$e,$f,$g,$h,$i,$j,$k,$l,$m,$n))
}
```

```
rule spyeye_plugins : banker
{
    meta:
        author = "Jean-Philippe Teissier / @Jipe_"
        description = "SpyEye X.Y Plugins memory"
        date = "2012-05-23"
        version = "1.0"
        filetype = "memory"

    strings:
        $a = "webfakes.dll"
        $b = "config.dat"           //may raise some FP
        $c = "collectors.txt"
        $d = "webinjects.txt"
        $e = "screenshots.txt"
        $f = "billinghammer.dll"
        $g = "block.dll"            //may raise some FP
        $h = "bugreport.dll"        //may raise some FP
        $i = "ccgrabber.dll"
        $j = "connector2.dll"
        $k = "creditgrab.dll"
        $l = "customconnector.dll"
        $m = "ffcertgrabber.dll"
        $n = "ftpbc.dll"
        $o = "rdp.dll"              //may raise some FP
        $p = "rt_2_4.dll"
        $q = "socks5.dll"           //may raise some FP
        $r = "spySpread.dll"
        $s = "w2chek4_4.dll"
        $t = "w2chek4_6.dll"

    condition:
        any of them
}
```

Source: https://github.com/jipegit/yara-rules-public

Binary Obfuscation

- Malware can use several techniques to **obfuscate** its binary code:
  - to make the comprehension harder
  - to prevent reverse engineering on a large-scale
  - to produce variants bypassing signature-based detection tools (e.g., antivirus).
- Techniques using **encryption**:
  - malware is composed of a decrypting stage and an encrypted body
  - by using **different keys** for each infection, malware body is changed
  - **metamorphic malware** and **mutation engines** allow to change the code of the malware and the cryptographic keys making each infection unique
  - mutations make the creation of unique/general signature(s) difficult.

Binary Obfuscation

- In general, **binary obfuscation** techniques are:
  - similar to those used in source code
  - applied at a binary level
  - various mechanisms, will briefly review the simpler here.
- **Dead-code insertion**:
  - useless code is inserted to make the analysis harder or to produce mutations
  - **example**: NOP operations are placed in the binary.
- **Register reassignment**:
  - various incarnations of malware use different set of registers
  - **example**: EAX, EBX, and EDX are swapped with EBX, EDX, and EAX.

Binary Obfuscation

- **Subroutine reordering**:
  - the order of subroutines are scrambled in a random manner.
- **Instruction substitution**:
  - an instruction (or a group) can be changed with an equivalent one
  - **example**: MOV can be replaced with PUSH/POP (but it is slower).
- **Code transposition**:
  - the sequence of instructions is rearranged
  - original functionalities are preserved via jumps (JMP) and branches
  - **example:** see the figure!

push: pushes (writes) a value to the stack.

pop: stores what is on the stack on a register.

push 02 (writes 02 on the stack)
pop eax (picks the value on top of the stack, i.e., 02 and writes it in eax)

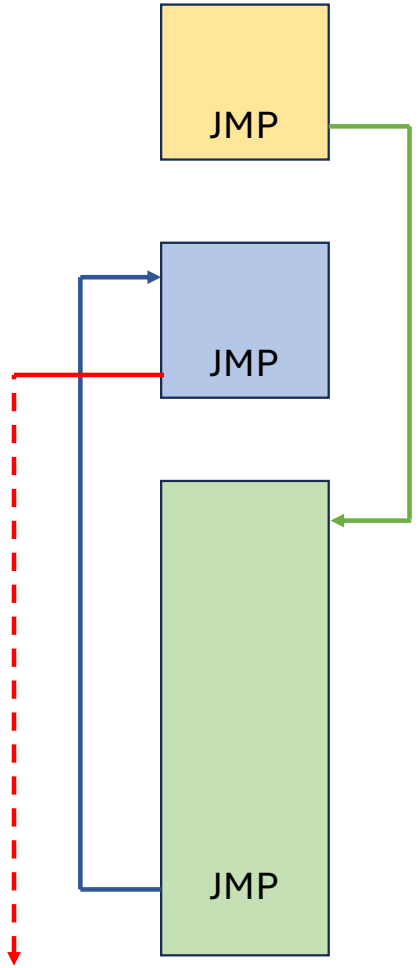
eax now contains 2

mov eax, 02 (copies 02 in eax)

eax now contains 2

Binary Obfuscation

- **Subroutine reordering**:
  - the order of subroutines are scrambled in a random manner.
- **Instruction substitution**:
  - an instruction (or a group) can be changed with an equivalent one
  - **example**: MOV can be replaced with PUSH/POP (but it is slower).
- **Code transposition**:
  - the sequence of instructions is rearranged
  - original functionalities are preserved via jumps (JMP) and branches
  - **example:** see the figure!

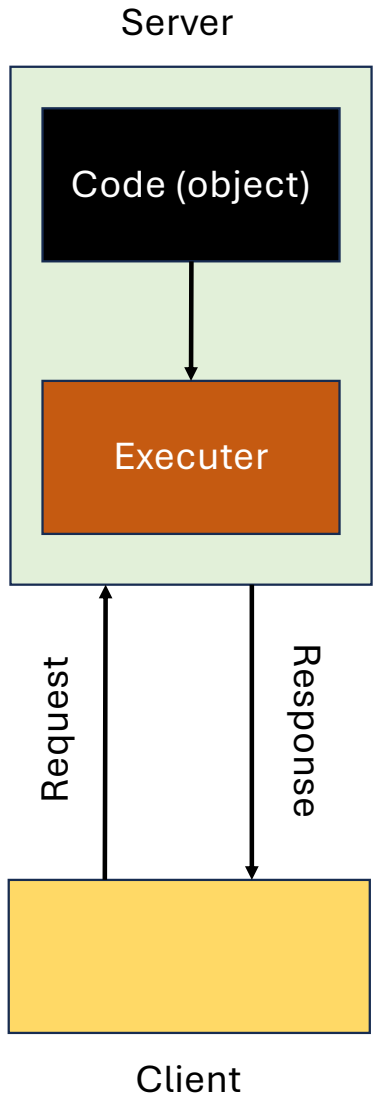


Source Code Obfuscation

- Malware can:
  - be (partially) implemented via non-compiled languages, e.g., Python
  - exploit scripting languages, e.g., PowerShell
  - live within platform-independent/JIT-compiled ecosystems, e.g., Java bytecode
  - target Web browsers, e.g., via JavaScript
  - have its source code leaked.
- The availability of source code may give defensive advantages:
  - understanding how the malicious software operate
  - making easier to identify and block attacks
  - improves the creation of signatures for detection.

Source Code Obfuscation

- Source code **obfuscation** tries to balance the (public) availability of source code.
- In essence, obfuscation:
  - **hinders** the **reverse engineering** process
  - rooted in a vast literature of intellectual property enforcement
  - can be defeated with enough time, dedication, and skills
  - can leverage automated tools.
- The best way to prevent reverse engineering is to not have nothing to reverse:
  - rely upon a server-side execution paradigm.



Source Code Obfuscation

- Source code **obfuscation** operates by:
  - **transforming** the original code in something **difficult to comprehend**
  - functionalities of the original code must be preserved
  - there is always a price to pay (e.g., in terms of performance).
- Source code obfuscation is a very vast field:
  - many techniques
  - competitions on who writes the most abstruse code (for fun)
  - interventions of the compiler especially when optimizing code.
- Obfuscation should not be considered a security practice:
  - **security through obscurity is not a good idea**.

Source Code Obfuscation

- As an example, let us **consider JavaScript obfuscation**:
  - ubiquitously used for Web applications
  - supported by almost any browser
  - 71% of malware written in JavaScript uses some form of obfuscation.
- Malicious software written in JavaScript:
  - spyware
  - ransomware
  - cryptominers
  - worms
- A vast collection of JavaScript Malware can be found here:
  - JavaScript Malware Collection: *https://github.com/HynekPetrak/javascript-malware-collection*

Source Code Obfuscation

- Collberg(*) characterizes code obfuscation via three metrics:
  - **potency**: *to what degree is a human reader confused?*
  - **resilience**: *how well are automatic deobfuscation attacks resisted?*
  - **cost**: how much overhead is added to the application?
- The three metrics are coupled.
- Let us review the main obfuscation mechanisms
  - examples consider JavaScript
  - many general ideas are language-agnostic.

```
void P() {
    int a = b;
}
```

```
void P() {
    int i,a = 0;
    for(i=0;i<b;i++) {
        a++;
    }
}
```

(*) C. Collberg, C. Thomborson, D. Low, “A Taxonomy of Obfuscating Transformations”, 1997.

Source Code Obfuscation

- **Control Flow Flattening**:
  - aims at increasing the structural complexity of a program by converting its original control flow into a unique flat construct
  - usually done by embedding all control structures within a large single switch-case statement, governed through a dispatching variable.

Source Code Obfuscation

```
const { exec } = require("child_process");
function runCommand(userInput) {
 exec("ls " + userInput, (error, stdout, stderr) => {
  if (error) {
   console.error(`Error: ${error}`);
   return;
  }
  console.log(stdout);
 });
}
```

obfuscation

```
const { exec } = require("child_process");
function runCommand(userInput) {
 let cmd, error, stdout, stderr, pc = 0;
 let junk = 0; // dead code

 while (true) {
  switch (pc) {
   case 0:
    cmd = "ls " + userInput;
    pc = 1;
    break;
   case 1:
    exec(cmd, (e, out, err) => {
     pc = 2;
     error = e;
     stdout = out;
     stderr = err;
     switch (pc) {
      case 2:
       if (error) {
        console.error("Error: " + error);
        return;
       }
       pc = 3;
       break;
      case 3:
       console.log(stdout);
       return;
     }
    });
    return;
   case 99:
    junk = Math.random() * 42; // dead code
    break;
  }
 }
}
```

Flattened control flow ruled by **pc**

Source Code Obfuscation

- Control Flow Flattening:
  - aims at increasing the structural complexity of a program by converting its original control flow into a unique flat construct
  - usually done by embedding all control structures within a large single switch-case statement, governed through a dispatching variable.
- **Dead Code Injection**:
  - increases the complexity of the execution flow by introducing a sequence of statements that are never executed.

Source Code Obfuscation

```
const { exec } = require("child_process");
function runCommand(userInput) {
 exec("ls " + userInput, (error, stdout, stderr) => {
  if (error) {
   console.error(`Error: ${error}`);
   return;
  }
  console.log(stdout);
 });
}
```

obfuscation

```
const { exec } = require("child_process");
function runCommand(userInput) {
 let cmd, error, stdout, stderr, pc = 0;
 let junk = 0; // dead code

 while (true) {
    switch (pc) {
     case 0:
      cmd = "ls " + userInput;
      pc = 1;
      break;
     case 1:
      exec(cmd, (e, out, err) => {
       pc = 2;
       error = e;
       stdout = out;
       stderr = err;
       switch (pc) {
        case 2:
         if (error) {
          console.error("Error: " + error);
          return;
         }
         pc = 3;
         break;
        case 3:
         console.log(stdout);
         return;
       }
      });
      return;
     case 99:
      junk = Math.random() * 42; // dead code
      break;
    }
  }
}
```

junk and the **random()** function are never executed

Source Code Obfuscation

- Control Flow Flattening:
  - aims at increasing the structural complexity of a program by converting its original control flow into a unique flat construct
  - usually done by embedding all control structures within a large single switch-case statement, governed through a dispatching variable.
- Dead Code Injection:
  - increases the complexity of the execution flow by introducing a sequence of statements that are never executed.
- **String Array Encoding**:
  - hinders static analysis by replacing string literals with references to elements in an array.

Source Code Obfuscation

```
1  const { exec } = require("child_process");
2  function runCommand(userInput) {
3   exec("ls " + userInput, (error, stdout, stderr) => {
4    if (error) {
5     console.error(`Error: ${error}`);
6     return;
7    }
8    console.log(stdout);
9   });
10 }
```

obfuscation

```
1  const _m = require("child_process")["exec"];
2  function _R1(_i) {
3   const _s = ["ls ", _i];
4   _m(_s[0] + _s[1], (e, o, x) => {
5    if (e) {
6     console["error"]("Error: " + e);
7     return;
8    }
9    console["log"](o);
10  });
11 }
```

The actual command **_i** is evaluated at “runtime”

Source Code Obfuscation

- **Identifier Renaming**:
  - is a lexical obfuscation method that replaces meaningful names of functions, variables, and parameters with non-descriptive ones.

Source Code Obfuscation

```
const { exec } = require("child_process");
function runCommand(userInput) {
 exec("ls " + userInput, (error, stdout, stderr) => {
  if (error) {
   console.error(`Error: ${error}`);
   return;
  }
  console.log(stdout);
 });
}
```

obfuscation

```
const _m = require("child_process")["exec"];
function _R1(_i) {
 const _s = ["ls ", _i];
 _m(_s[0] + _s[1], (e, o, x) =>  {
 if (e) {
  console["error"]("Error: " + e);
  return;
 }
 console["log"](o);
 });
}
```

To make comprehension harder, identifiers are renamed, e.g., **userInput** is changed to **_i**

Source Code Obfuscation

- Identifier Renaming:
  - is a lexical obfuscation method that replaces meaningful names of functions, variables, and parameters with non-descriptive ones.
- **Opaque Predicates**:
  - are conditional expressions that always evaluate to a predetermined boolean value difficult to forecast without running the program
  - they do not alter the actual execution path, but they introduce redundant conditional branches that analysts and automated tools must process
  - often support the isolation of dead code.

Source Code Obfuscation

```
1  const { exec } = require("child_process");
2  function runCommand(userInput) {
3   exec("ls " + userInput, (error, stdout, stderr) => {
4    if (error) {
5     console.error(`Error: ${error}`);
6     return;
7    }
8    console.log(stdout);
9   });
10 }
```

obfuscation

```
1  const _m = require("child_process")["exec"];
2  function _R1(_i) {
3   const _s = ["ls ", _i];
4   _m(_s[0] + _s[1], (e, o, x) =>  {
5   if (e) {
6    console["error"]("Error: " + e);
7    return;
8   }
9   console["log"](o);
10  });
11 }
```

CWE-078: OS Command Injection

Source Code Obfuscation

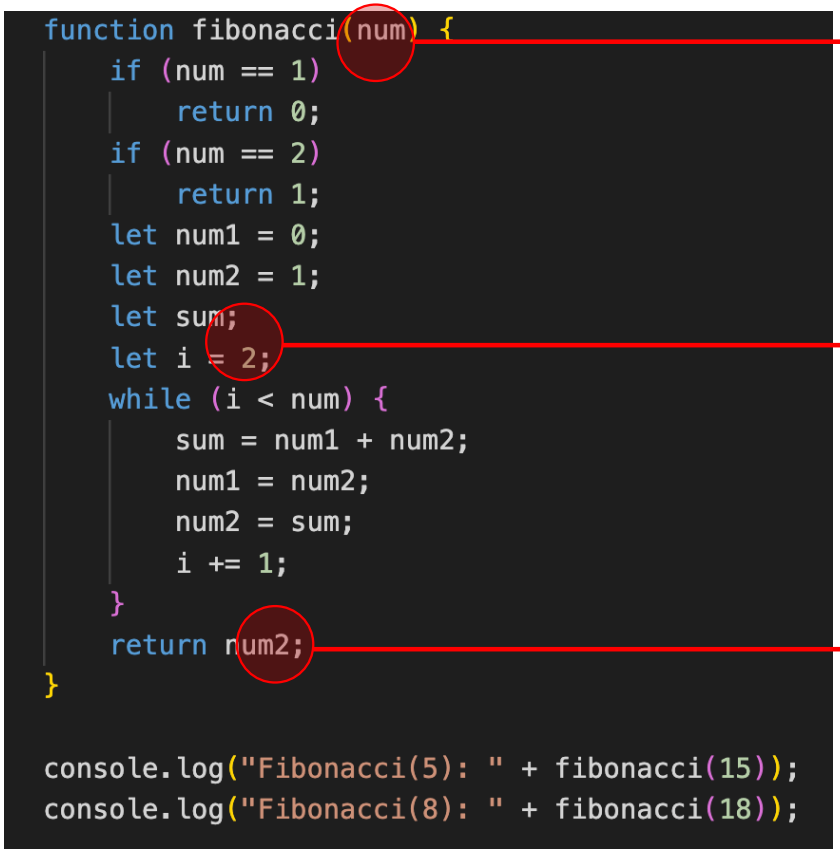
- Two **tools** to experiment with.
- Obfuscator:
  - **javascript-obfuscator** (*https://github.com/javascript-obfuscator/javascript-obfuscator*)
  - *npm install javascript-obfuscator*
- De-Obfuscator:
  - **js-deobfuscato**r (https://github.com/ben-sb/javascript-deobfuscator)
  - *npm install js-deobfuscator*
- Example:
  - Fibonacci.js: a simple legitimate .js for computing the Fibonacci sequence
  - borrowed from*: https://www.geeksforgeeks.org/*

Source Code Obfuscation

- javascript-deobfuscator offers a great control on the obfuscation process:
  - *javascript-deobfuscator --help* for more information.
- The first example is done by using the following configuration:
  - --compact false
  - --control-flow-flattening false
  - --string-array false
  - --simplify false
  - --dead-code-injection false
  - --rename-globals false
  - --log true
- Try to run the code with **node**.

Source Code Obfuscation

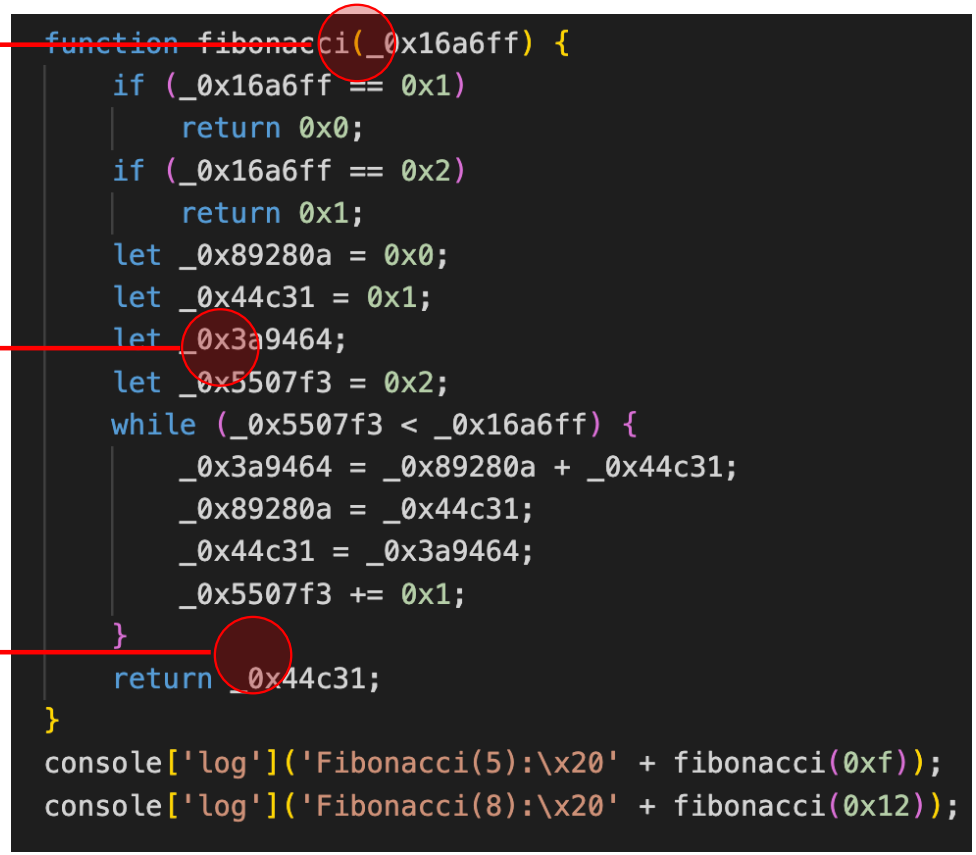
...various renaming



```
function fibonacci(num) {
    if (num == 1)
        return 0;
    if (num == 2)
        return 1;
    let num1 = 0;
    let num2 = 1;
    let sum;
    let i = 2;
    while (i < num) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
        i += 1;
    }
    return num2;
}

console.log("Fibonacci(5): " + fibonacci(15));
console.log("Fibonacci(8): " + fibonacci(18));
```

Fibonacci.js



```
function fibonacci(_0x16a6ff) {
    if (_0x16a6ff == 0x1)
        return 0x0;
    if (_0x16a6ff == 0x2)
        return 0x1;
    let _0x89280a = 0x0;
    let _0x44c31 = 0x1;
    let _0x3a9464;
    let _0x5507f3 = 0x2;
    while (_0x5507f3 < _0x16a6ff) {
        _0x3a9464 = _0x89280a + _0x44c31;
        _0x89280a = _0x44c31;
        _0x44c31 = _0x3a9464;
        _0x5507f3 += 0x1;
    }
    return _0x44c31;
}
console['log']('Fibonacci(5):\x20' + fibonacci(0xf));
console['log']('Fibonacci(8):\x20' + fibonacci(0x12));
```

Fibonacci-obfuscated-simple.js

Source Code Obfuscation

- The second example is done by using the following configuration:
  - --compact false
  - --control-flow-flattening true
  - --string-array false
  - --simplify false
  - --dead-code-injection false
  - --rename-globals false
  - --log true

Source Code Obfuscation

...attempt to burden the first part

```
function fibonacci(num) {
    if (num == 1)
        return 0;
    if (num == 2)
        return 1;
    let num1 = 0;
    let num2 = 1;
    let sum;
    let i = 2;
    while (i < num) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
        i += 1;
    }
    return num2;
}

console.log("Fibonacci(5): " + fibonacci(15));
console.log("Fibonacci(8): " + fibonacci(18));
```

Fibonacci.js

```
function fibonacci(_0x59b4f0) {
    const _0x8313b4 = {
        'GXrPe': function (_0x5eb877, _0x4a2d74) {
            return _0x5eb877 == _0x4a2d74;
        },
        'lJaTf': function (_0x358f2c, _0x5e2b46) {
            return _0x358f2c == _0x5e2b46;
        },
        'JoTzM': function (_0x36b21f, _0x352200) {
            return _0x36b21f < _0x352200;
        }
    };
    if (_0x8313b4['GXrPe'](_0x59b4f0, 0x1))
        return 0x0;
    if (_0x8313b4['lJaTf'](_0x59b4f0, 0x2))
        return 0x1;
    let _0x336463 = 0x0;
    let _0x57f830 = 0x1;
    let _0x2866d1;
    let _0xa24eb3 = 0x2;
    while (_0x8313b4['JoTzM'](_0xa24eb3, _0x59b4f0)) {
        _0x2866d1 = _0x336463 + _0x57f830;
        _0x336463 = _0x57f830;
        _0x57f830 = _0x2866d1;
        _0xa24eb3 = 0x1;
    }
    return _0x57f830;
}
console['log']('Fibonacci(5):\x20' + fibonacci(0xf));
console['log']('Fibonacci(8):\x20' + fibonacci(0x12));
```

let _0x57f830: number

Fibonacci-obfuscated-flat.js

Source Code Obfuscation

- The third example is done by using the following configuration:
  - --compact false
  - --control-flow-flattening true
  - --string-array true
  - --simplify true
  - --dead-code-injection true
  - --rename-globals true
  - --log true
- Hard to report:
  - results in *Fibonacci-obfuscated-various.js*

Source Code Obfuscation

- The last example is done by using the following configuration:
  - --compact true
  - --control-flow-flattening true
  - --string-array true
  - --simplify true
  - --dead-code-injection true
  - --rename-globals true
  - --log true

Source Code Obfuscation

```
function fibonacci(num) {
    if (num == 1)
        return 0;
    if (num == 2)
        return 1;
    let num1 = 0;
    let num2 = 1;
    let sum;
    let i = 2;
    while (i < num) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
        i += 1;
    }
    return num2;
}

console.log("Fibonacci(5): " + fibonacci(15));
console.log("Fibonacci(8): " + fibonacci(18));
```

Fibonacci.js

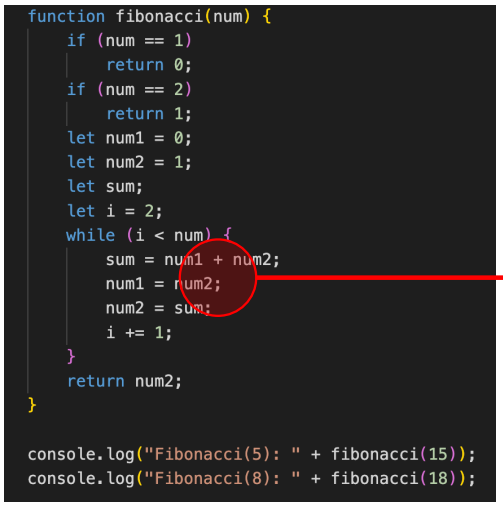
...obfuscated and indentation removed to fit one line

```
const a0_0x2d4c6a=a0_0x26d1;(function(_0x1aeeb5,_0x907d3){const
_0x3096b0=a0_0x26d1,_0x6bc6e=_0x1aeeb5();while(!![]){try{const
_0x79283a=parseInt(_0x3096b0(0x1f9))/0x1*(-parseInt(_0x3096b0(0x1ee))/0x2)
+-parseInt(_0x3096b0(0x1ef))/0x3*(parseInt(_0x3096b0(0x1f4))/0x4)+-parseInt
(_0x3096b0(0x1f6))/0x5*(-parseInt(_0x3096b0(0x1fc))/0x6)+parseInt(_0x3096b0
(0x1f2))/0x7+-parseInt(_0x3096b0(0x1ec))/0x8*(-parseInt(_0x3096b0(0x1f3))/
0x9)+parseInt(_0x3096b0(0x1ed))/0xa*(-parseInt(_0x3096b0(0x1f0))/0xb)
+parseInt(_0x3096b0(0x1eb))/0xc;if(_0x79283a===_0x907d3)break;else _0x6bc6e
['push'](_0x6bc6e['shift']());}catch(_0x51ad37){_0x6bc6e['push'](_0x6bc6e
['shift']());}}}(a0_0x16f2,0xcd775));function a0_0x157615(_0x50d2ac){const
_0x1b5fb1=a0_0x26d1,_0x4b0ebb={'pVsWD':function(_0x27ed9d,_0x54c5e1){return
_0x27ed9d==_0x54c5e1;},'YWHRt':function(_0x123047,_0x4bec46){return
_0x123047<_0x4bec46;},'vUBfX':function(_0x3bc39e,_0x4b736c){return _0x3bc39e
+_0x4b736c;}};if(_0x4b0ebb[_0x1b5fb1(0x1f7)](_0x50d2ac,0x1))return 0x0;if
(_0x50d2ac==0x2)return 0x1;let _0x3805f4=0x0,_0x263d06=0x1,_0x2d73d1,
_0x462335=0x2;while(_0x4b0ebb[_0x1b5fb1(0x1f5)](_0x462335,_0x50d2ac))
{_0x2d73d1=_0x4b0ebb[_0x1b5fb1(0x1f8)](_0x3805f4,_0x263d06),
_0x3805f4=_0x263d06,_0x263d06=_0x2d73d1,_0x462335+=0x1;}return _0x263d06;}
console['log'](a0_0x2d4c6a(0x1fb)+a0_0x157615(0xf)),console[a0_0x2d4c6a
(0x1fa)](a0_0x2d4c6a(0x1f1)+a0_0x157615(0x12));function a0_0x26d1(_0x5ea56c,
_0x279fac){const _0x16f264=a0_0x16f2();return a0_0x26d1=function(_0x26d129,
_0x2163ae){_0x26d129=_0x26d129-0x1eb;let _0x406a6d=_0x16f264[_0x26d129];
return _0x406a6d;},a0_0x26d1(_0x5ea56c,_0x279fac);}function a0_0x16f2()
{const _0x2fa373=['vUBfX','4MwgwBr','log','Fibonacci(5):\x20','36822EsiFnd',
'16266168BvBbtU','78654080AfcDl','55160NjLeRn','488588GZZlHR',
'4960281glLsAP','99EgUBpv','Fibonacci(8):\x20','4544589UvqWby','9tYcKdq',
'4MCJyjS','YWHRt','435owIXHO','pVsWD'];a0_0x16f2=function(){return
_0x2fa373;};return a0_0x16f2();}
```

Fibonacci-obfuscated-compact.js

Source Code Obfuscation

- Try to **de-obfuscate** two example scripts:
  - *js-deobfuscator -i Fibonacci-obfuscated-flat.js -o Fibonacci-deobfuscated.js*
  - *js-deobfuscator -i Fibonacci-obfuscated-compact.js -o Fibonacci-deobfuscated-bis.js*



```
function fibonacci(num) {
    if (num == 1)
        return 0;
    if (num == 2)
        return 1;
    let num1 = 0;
    let num2 = 1;
    let sum;
    let i = 2;
    while (i < num) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
        i += 1;
    }
    return num2;
}

console.log("Fibonacci(5): " + fibonacci(15));
console.log("Fibonacci(8): " + fibonacci(18));
```

Fibonacci.js

```
function fibonacci(_0x59b4f0) {
  const _0x8313b4 = {GXrPe: function (_0x5eb877, _0x4a2d74) {
    return _0x5eb877 == _0x4a2d74;
  }, lJaTf: function (_0x358f2c, _0x5e2b46) {
    return _0x358f2c == _0x5e2b46;
  }, JoTzM: function (_0x36b21f, _0x352200) {
    return _0x36b21f < _0x352200;
  }};
  if (_0x8313b4.GXrPe(_0x59b4f0, 1)) return 0;
  if (_0x8313b4.lJaTf(_0x59b4f0, 2)) return 1;
  let _0x336463 = 0;
  let _0x57f830 = 1;
  let _0x2866d1;
  let _0xa24eb3 = 2;
  while (_0x8313b4.JoTzM(_0xa24eb3, _0x59b4f0)) {
    _0x2866d1 = _0x336463 + _0x57f830;
    _0x336463 = _0x57f830;
    _0x57f830 = _0x2866d1;
    _0xa24eb3 += 1;
  }
  return _0x57f830;
}
console.log("Fibonacci(5): " + fibonacci(15));
console.log("Fibonacci(8): " + fibonacci(18));
```

Fibonacci-deobfuscated.js

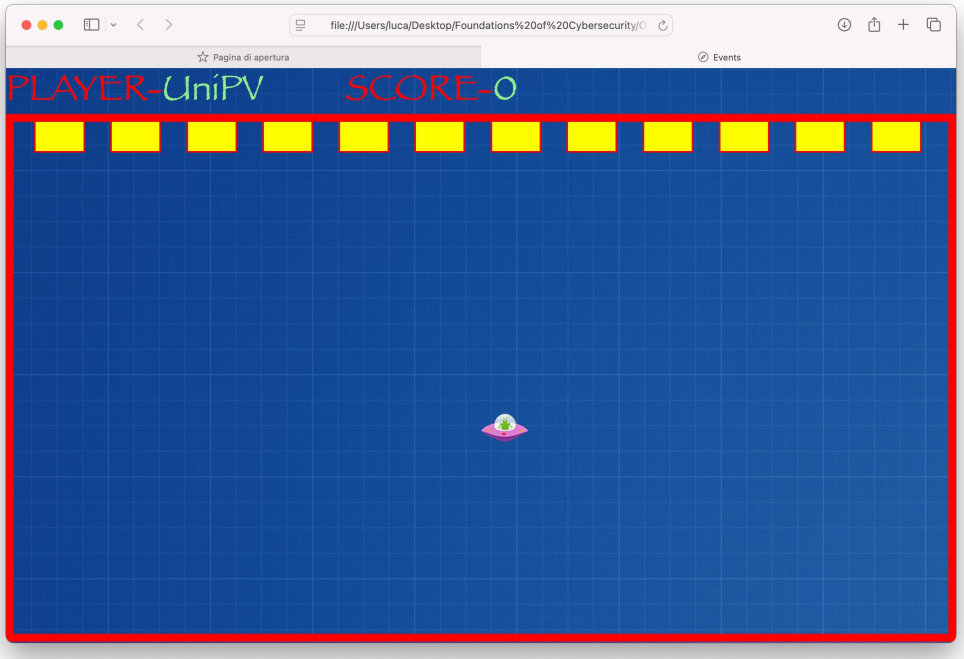
Notice: removing identifiers is a **one-way** transformation: once the original names are gone, they cannot be recovered.

Source Code Obfuscation

- As a second example let us consider a slightly more complex program:
  - DoodleShip implementation by Naklecha
  - GitHub: https://github.com/naklecha/doodleship
  - implemented in HTML, CSS, and JavaScript
  - let us give the guy some stars :)
  - not in the repo of the course, download from Neklecha.
- Obfuscation done via:
  - javascript-obfuscator --options-preset high-obfuscation --simplify true --self-defending true jssp.js

Source Code Obfuscation

```
var a0_0x407e5f=a0_0x1adc;(function(_0x2863bd,_0xe8513a){var _0x275c15=a0_0x1adc,_0x23ab97=_0x2863bd();while(!![]){try{var
_0x2c4f89=parseInt(_0x275c15(0x1bd))/0x1*(-parseInt(_0x275c15(0x1cf))/0x2)+parseInt(_0x275c15(0x1c7))/0x3*(-parseInt(_0x275c15
(0x1c4))/0x4)+parseInt(_0x275c15(0x1d1))/0x5+-parseInt(_0x275c15(0x1bc))/0x6*(-parseInt(_0x275c15(0x1c0))/0x7)+parseInt(_0x275c15
(0x1c8))/0x8+-parseInt(_0x275c15(0x1cb))/0x9*(parseInt(_0x275c15(0x1c9))/0xa)+parseInt(_0x275c15(0x1d3))/0xb;if
(_0x2c4f89===_0xe8513a)break;else _0x23ab97['push'](_0x23ab97['shift']());}catch(_0x3ce054){_0x23ab97['push'](_0x23ab97['shift']
());}}}(a0_0x3aa6,0x40746));var a0_0x23883d=(function(){var _0x3675e5=!![];return function(_0x46a3df,_0x255fa5){var
_0x909be1=_0x3675e5?function(){var _0x476a81=a0_0x1adc;if(_0x255fa5){var _0x27578c=_0x255fa5[_0x476a81(0x1d2)](_0x46a3df,arguments);
return _0x255fa5=null,_0x27578c;}}:function(){};return _0x3675e5=!![],_0x909be1;};})(),a0_0x24d8a3=a0_0x23883d(this,function(){var
_0x597ab0=a0_0x1adc;return a0_0x24d8a3[_0x597ab0(0x1be)]()['search']('(((.+)+)+)+$')['toString']()[_0x597ab0(0x1d6)](a0_0x24d8a3)
['search']('(((.+)+)+)+$');});function a0_0x3aa6(){var _0x3cd4fc=['1368055rCaDSN','apply','4425333anylkD',
'Enter\x20your\x20name:\x20','keypress','constructor','innerHTML','6Ovjebn','83QNmLHX','toString','top','1970899bJCcuV','.robo>div',
'#score','toLowerCase','145364AZLcQH','style','left','9CEUupG','3508184NnXtvX','5125150ZXaSgR','reset','9ipsvnh','#player',
'querySelector','#name','12300vvGkeA','length'];a0_0x3aa6=function(){return _0x3cd4fc;};return a0_0x3aa6();}a0_0x24d8a3();var obj={};
function a0_0x1adc(_0x429160,_0x333864){var _0x1e120a=a0_0x3aa6();return a0_0x1adc=function(_0x24d8a3,_0x23883d)
{_0x24d8a3=_0x24d8a3-0x1bc;var _0x3aa6cd=_0x1e120a[_0x24d8a3];return _0x3aa6cd;},a0_0x1adc(_0x429160,_0x333864);}obj['reset']
=function(){var _0xbfe7c6=a0_0x1adc;alert(document['querySelector']('#name')[_0xbfe7c6(0x1d7)]+',\x20sorry!\x20You\x20DIED!');for
(i=0x0;i<robots['length'];i++){robots[i]['style'][_0xbfe7c6(0x1bf)]=0x9+'vh',curr[i]=0x9;}d=0x3c,l=0x32,spaceship[_0xbfe7c6(0x1c5)]
[_0xbfe7c6(0x1bf)]=d+'vh',spaceship['style']['left']=l+'vw',document[_0xbfe7c6(0x1cd)]('#score')[_0xbfe7c6(0x1d7)]=0x0,document
[_0xbfe7c6(0x1cd)](_0xbfe7c6(0x1ce))[_0xbfe7c6(0x1d7)]=prompt('Enter\x20new\x20player\x20name:\x20');},obj['f']=function(){var
_0x3c248b=a0_0x1adc;a=parseInt(spaceship[_0x3c248b(0x1c5)][_0x3c248b(0x1c6)]),b=parseInt(spaceship[_0x3c248b(0x1c5)][_0x3c248b
(0x1bf)]);for(i=0x0;i<robots['length'];i++){if(parseInt(robots[i]['style'][_0x3c248b(0x1c6)])+0x5>a&&a>parseInt(robots[i][_0x3c248b
(0x1c5)][_0x3c248b(0x1c6)])||a+0x5>parseInt(robots[i][_0x3c248b(0x1c5)][_0x3c248b(0x1c6)])&&a<parseInt(robots[i][_0x3c248b(0x1c5)]
['left'])||a+0x5<=parseInt(robots[i]['style'][_0x3c248b(0x1c6)])+0x5&&a>=parseInt(robots[i][_0x3c248b(0x1c5)][_0x3c248b(0x1c6)])){if
(parseInt(robots[i][_0x3c248b(0x1c5)]['top'])+0x5>b&&b+0x5>parseInt(robots[i][_0x3c248b(0x1c5)][_0x3c248b(0x1bf)])){obj[_0x3c248b
(0x1ca)]();return;}}}k=~~event['key'][_0x3c248b(0x1c3)]();if(k=='s'){if(0x64<=d+0x7){obj[_0x3c248b(0x1ca)]();return;}d+=0x2,spaceship
['style'][_0x3c248b(0x1bf)]=d+'vh';}else{if(k=='w'){if(d<=0x8){obj[_0x3c248b(0x1ca)]();return;}d-=0x2,spaceship[_0x3c248b(0x1c5)]
[_0x3c248b(0x1bf)]=d+'vh';}else{if(k=='d'){if(0x64<=l+0x6){obj['reset']();return;}l+=0x2,spaceship[_0x3c248b(0x1c5)][_0x3c248b
(0x1c6)]=l+'vw';}else{if(k=='a'){if(l<=0x0){obj[_0x3c248b(0x1ca)]();return;}l-=0x2,spaceship[_0x3c248b(0x1c5)][_0x3c248b(0x1c6)]=l
+'vw';}}}}a=parseInt(spaceship[_0x3c248b(0x1c5)]['left']),b=parseInt(spaceship[_0x3c248b(0x1c5)][_0x3c248b(0x1bf)]);for(i=0x0;
i<robots[_0x3c248b(0x1d0)];i++){robots[i]['style']['top']=curr[i]+'vh',curr[i]=curr[i]+speeds[i],curr[i]>=0x5e&&(curr[i]=0x9,k=Math
['random']()*0x7,k<0x1&&(k=0x1),speeds[i]=k,document[_0x3c248b(0x1cd)](_0x3c248b(0x1c2))[_0x3c248b(0x1d7)]=parseInt(document
[_0x3c248b(0x1cd)](_0x3c248b(0x1c2))[_0x3c248b(0x1d7)])+0x1);}},document[a0_0x407e5f(0x1cd)]('#name')[a0_0x407e5f(0x1d7)]=prompt
(a0_0x407e5f(0x1d4)),d=0x3c,l=0x32,robots=document['querySelectorAll'](a0_0x407e5f(0x1c1)),speeds=[];for(i=0x0;i<robots['length'];i
++){k=Math['random']()**0x2*0x7,k<0x1&&(k=0x1),speeds[i]=k;}curr=[];for(i=0x0;i<robots[a0_0x407e5f(0x1d0)];i++){curr[i]=speeds[i]
+0x9;}spaceship=document[a0_0x407e5f(0x1cd)](a0_0x407e5f(0x1cc)),window['addEventListener'](a0_0x407e5f(0x1d5),obj['f'],![]);
```



JavaScript Minification

- The process of **minification** (or **minimization**) of JavaScript code allows to:
  - remove unnecessary characters, e.g., whitespace, comments, and semicolons
  - shorten variable names and functions
  - not altering the code functionality.
- Major benefits of minification:
  - **reduced bandwidth utilization**: less data, less money (especially for high-traffic sites)
  - **shorter page loading times**: smaller pages lead to quicker results (useful for mobile devices)
  - **improved user experience**: sites are snappier and "promoted" by some search engines.
- Limitations and issues of the minification:
  - if by mistake authentication tokens or API keys are left in the code, they will be preserved
  - without comments, **code is harder to maintain and understand**
  - it is not a way to secure code
  - being a basic re-formatting process, it is unlikely that it introduces new vulnerabilities.
- Attackers use minification as another layer of complexity when analyzing code.

JavaScript Minification

- Two **minifiers** to experiment with:
  - **Minify** (*https://github.com/coderaiser/minify*)
  - **Minify** (*https://github.com/matthiasmullie/minify*)
  - they both have the same name 😅
- Examples have been done with the Minify of coderaiser:
  - *install npm i minify -g*
  - *minify target*
- Sample One:
  - ChartExample.html and GeoLocation.html: two simple legitimate .js files
  - borrowed from W3CSchools (*https://www.w3schools.com)*.

JavaScript Minification

- Sample One:
  - ChartExample.html and GeoLocation.html: two simple legitimate .js files
  - borrowed from W3CSchools (*https://www.w3schools.com*).

```
<!DOCTYPE html>
<html>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<body>

<div id="myPlot" style="width:100%;max-width:700px"></div>

<script>
const xArray = [55, 49, 44, 24, 15];
const yArray = ["Italy ", "France ", "Spain ", "USA ", "Argentina "];

const data = [{
  x:xArray,
  y:yArray,
  type:"bar",
  orientation:"h",
  marker: {color:"rgba(255,0,0,0.6)"}
}];

const layout = {title:"World Wide Wine Production"};

Plotly.newPlot("myPlot", data, layout);
</script>

</body>
</html>
```

ChartExample.html

minification
(from 509 Bytes to 409 Bytes)

```
<!doctype html><script src=https://cdn.plot.ly/plotly-latest.min.js></script><div id=myPlot style=width:100%;
max-width:700px></div><script>const xArray=[55,49,44,24,15],yArray=["Italy ","France ","Spain ","USA ",
"Argentina "],data=[{x:xArray,y:yArray,type:"bar",orientation:"h",marker:{color:"rgba(255,0,0,0.6)"}}],
layout={title:"World Wide Wine Production"};Plotly.newPlot("myPlot",data,layout)</script>
```

ChartExampleMinified.html

JavaScript Minification

- Sample Two:
  - 20170110_9330ee612a9027120543d6cd601cda83 is a malicious .js
  - borrowed from: JavaScript Malware Collection (*https://github.com/HynekPetrak/javascript-malware-collection*)
  - originally obfuscated(*) has been further minified
  - both original and minified versions have been “disarmed” with some comments.
- Try to:
  - read it
  - *du -sh 2017[...]* to evaluate the reduction in terms of size.

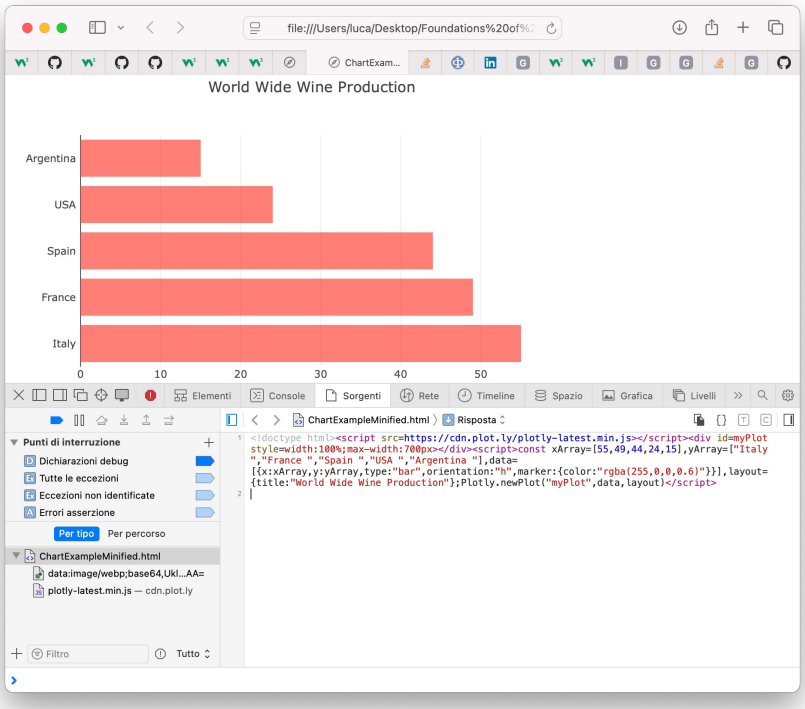
(*) Analysis: A. Herrera, “Optimizing Away JavaScript Obfuscation”, *IEEE 20th International Working Conference on Source Code Analysis and Manipulation, Oct.* 2020, pp. 215-220

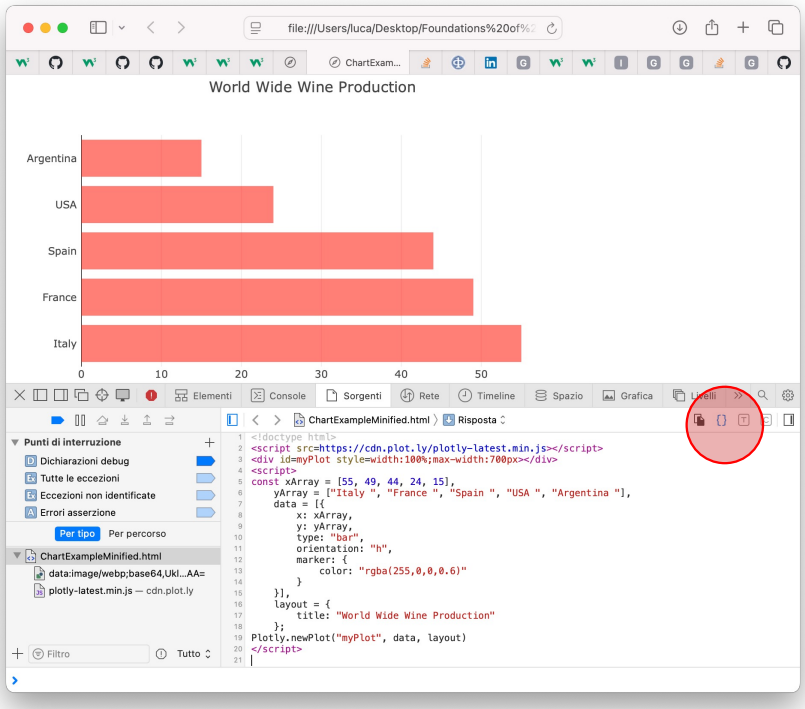
JavaScript Minification

- The minification process can be (almost) reversed:
  - the JavaScript when through **unminifying**.
- A possible tool is:
  - **Unminify** (*https://github.com/shapesecurity/unminify*)
  - *npm install -g unminify*
  - *unminify target*
  - compare the original and unminifed .js
  - a bit of processing is needed if the JavaScript is embedded in HTML.

JavaScript Minification

- Sone browsers can “beutify” code and (almost) revert minification.





Wrap Up

- **Malware** is an **umbrella** term for a wide-array of **malicious software**.
- Malicious software is usually composed of recurrent **stages**, which could be also **remotely** retrieved.
- Understanding the behavior of malware requires **different types** of **investigations**, i.e., **static** and **dynamic** analysis.
- Several **tools** are available and **we just scratched the surface**.
- The **identification** and **classification** of malware can also rely upon “standard” tools, for instance **YARA rules**.
- Alas, **countermeasures** trigger an **arms race** and attackers may use **packers** and various **obfuscation** techniques.