Co-funded by the
Erasmus+ Programme
of the European Union

ISSES – Information Security Services
Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant
N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

REVERSE ENGINEERING

Serbian Cybersecurity Challenge 2020

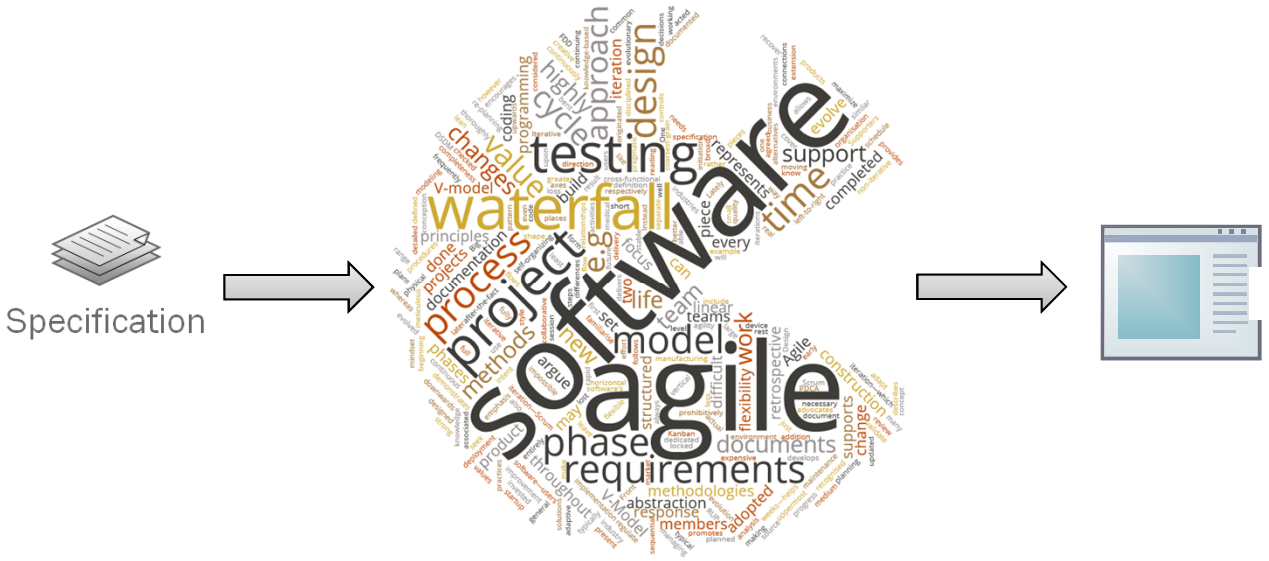
ISSES 2017-2020
Novi Sad, Serbia

Vision

- From Zero to Hero in reverse engineering
- Malware analysis
  - Static
  - Dynamic
- Reversing custom programs
- Portable Executable (PE)
- Executable and Linkable Format (ELF)

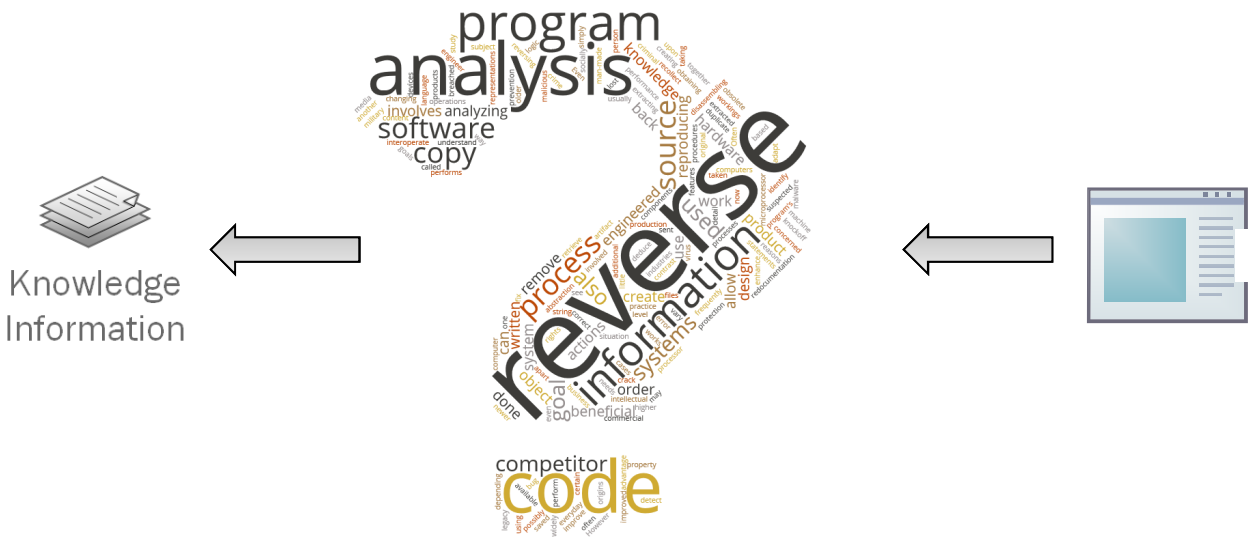
Reverse Engineering

- Starting point: specification
- Goal: code that confirms with the sepcification



Reverse Engineering

- Starting point: application
- Goal: extract knowledge or design information



- Bug finding, interfacing, intelligence gathering about competitors

PORTABLE EXECUTABLE (PE)

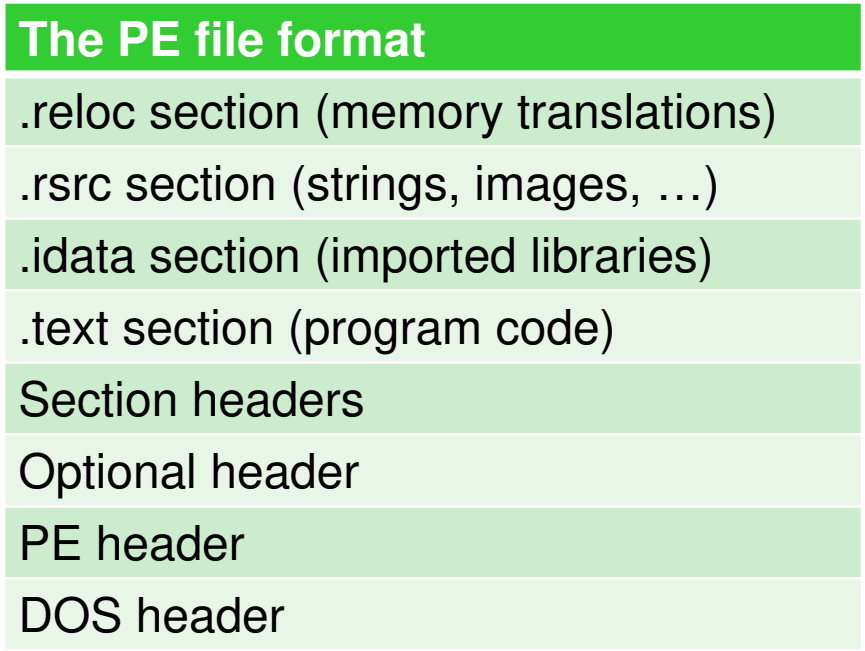
Windows OS: Portable Executable (PE)

- The Portable Executable (PE) format is a file format for executables, object code and DLLs used in the 32-bit and 64-bit Windows OS.
- “Portable” refers to the format’s versatility in numerous environments of OS architecture.

The PE file format
.reloc section (memory translations)
.rsrc section (strings, images, …)
.idata section (imported libraries)
.text section (program code)
Section headers
Optional header
PE header
DOS header

PE structure – 1

- **DOS header:** Starts with 'ZB', i.e. Mark Zbikowsky, one of the original DOS architects. Last 4 bytes point to the PE header.
- **PE header:** Contains general attributes: program defined for 32 or 64-bit, compilation timestamp (when was the PE file built), etc.
- **Optional header:** Defines the program's entry point (1st instruction), data size loaded by Windows, program target (GUI vs cmd), etc.
- **Section headers:** Describe the sections contained in the PE file. Permissions granted to sections.
  - NOTE: A section is a chunk of data or instructions loaded by Windows.



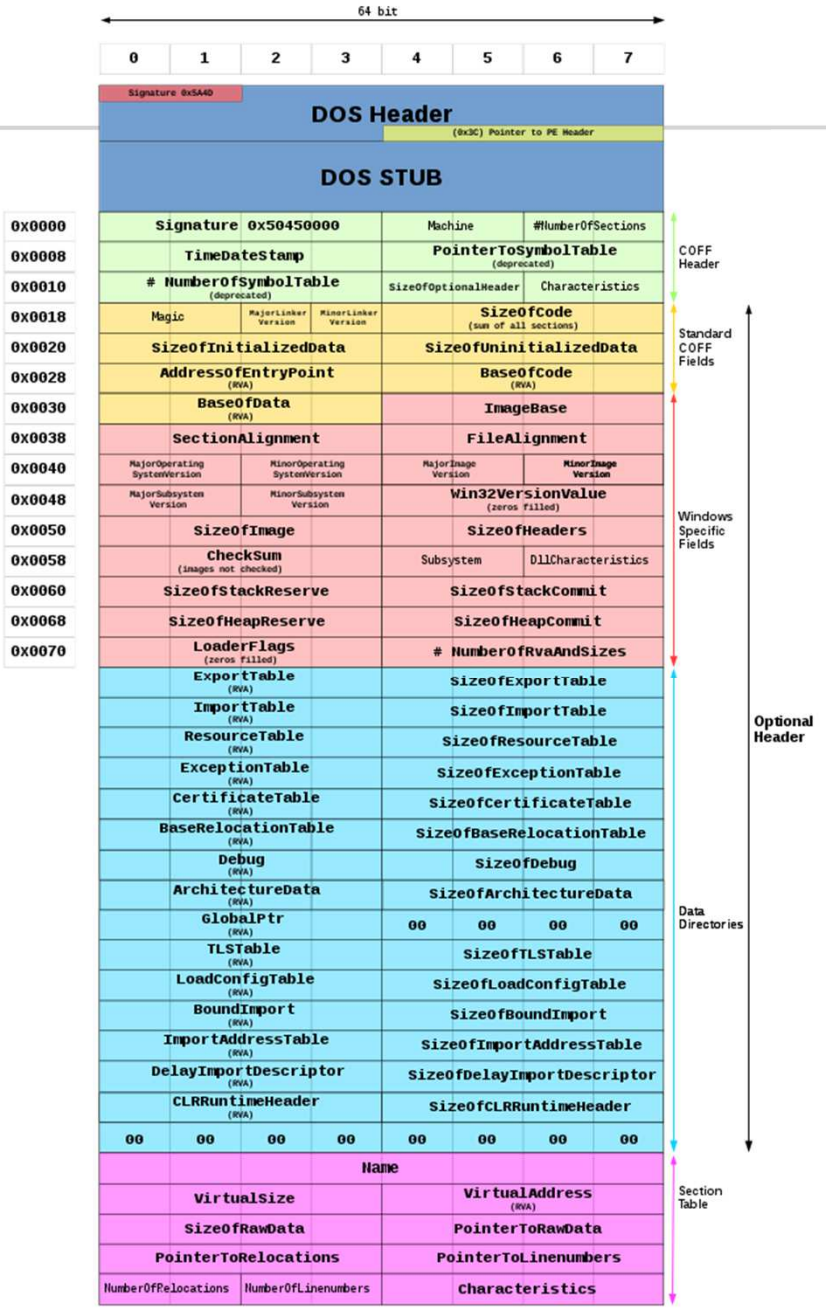
PE structure – 2

- **.text section:** The x86 code. Each PE file has at least one such section.
- **.idata section:** Aka "imports", contains the Import Address Table, list of loaded dynamically linked libraries (DLLs) and their functions.
- **Data sections:** .rsrc, .data & .rdata store cursor images, button skins, audio and other necessary elements.
- **.reloc section:** allows the code to be moved around in memory. Defines memory address translations for Windows.

The PE file format
.reloc section (memory translations)
.rsrc section (strings, images, …)
.idata section (imported libraries)
.text section (program code)
Section headers
Optional header
PE header
DOS header

Portable Executable

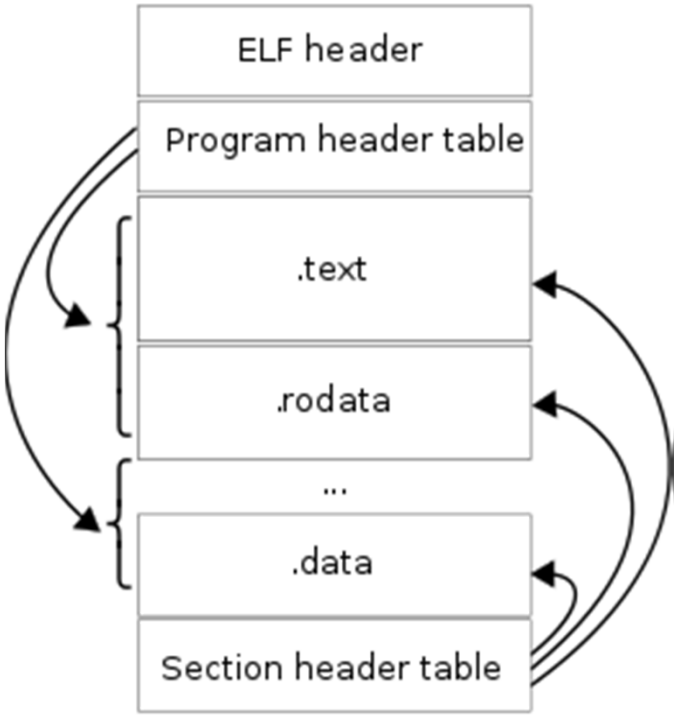
CC BY 4.0,
https://commons.wikimedia.org/w/index.php?curid=51026079



Executable and Linkable Format (ELF)

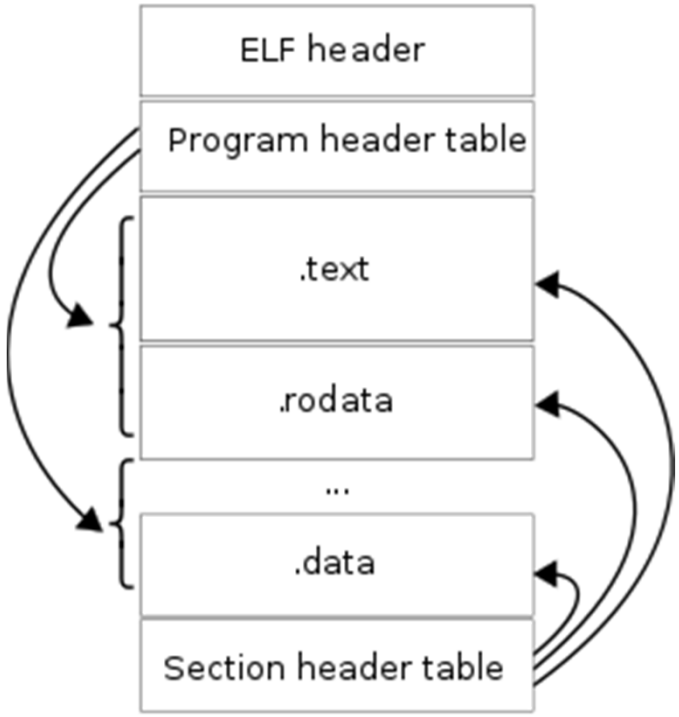
ELF file structure – 1

- **ELF header:** 32- or 64-bit addresses.
- **Program header table:** Tells the (*nix) system how to create a process image. Describes zero or more memory segments.
- **Section header table:** Describes zero or more sections.
- Data referred to by entries in the program header table or section header table
- Trivia: also used by Android, Sony PlayStation 2-4.



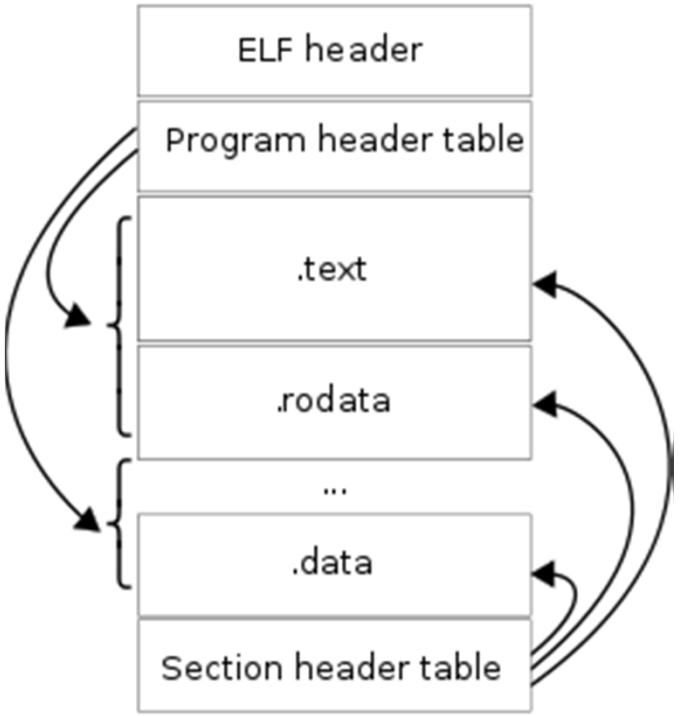
ELF file structure – 3

- **.text:** Executable instructions of the program.
- **.data:** Initialized data.
- **.rodata:** Read-only data.
- **.bss:** Un-initialized data. The system inits these with zeros.
- **.rel.text, rel.data, rel.rodata:** relocation information for the corresponding sections.

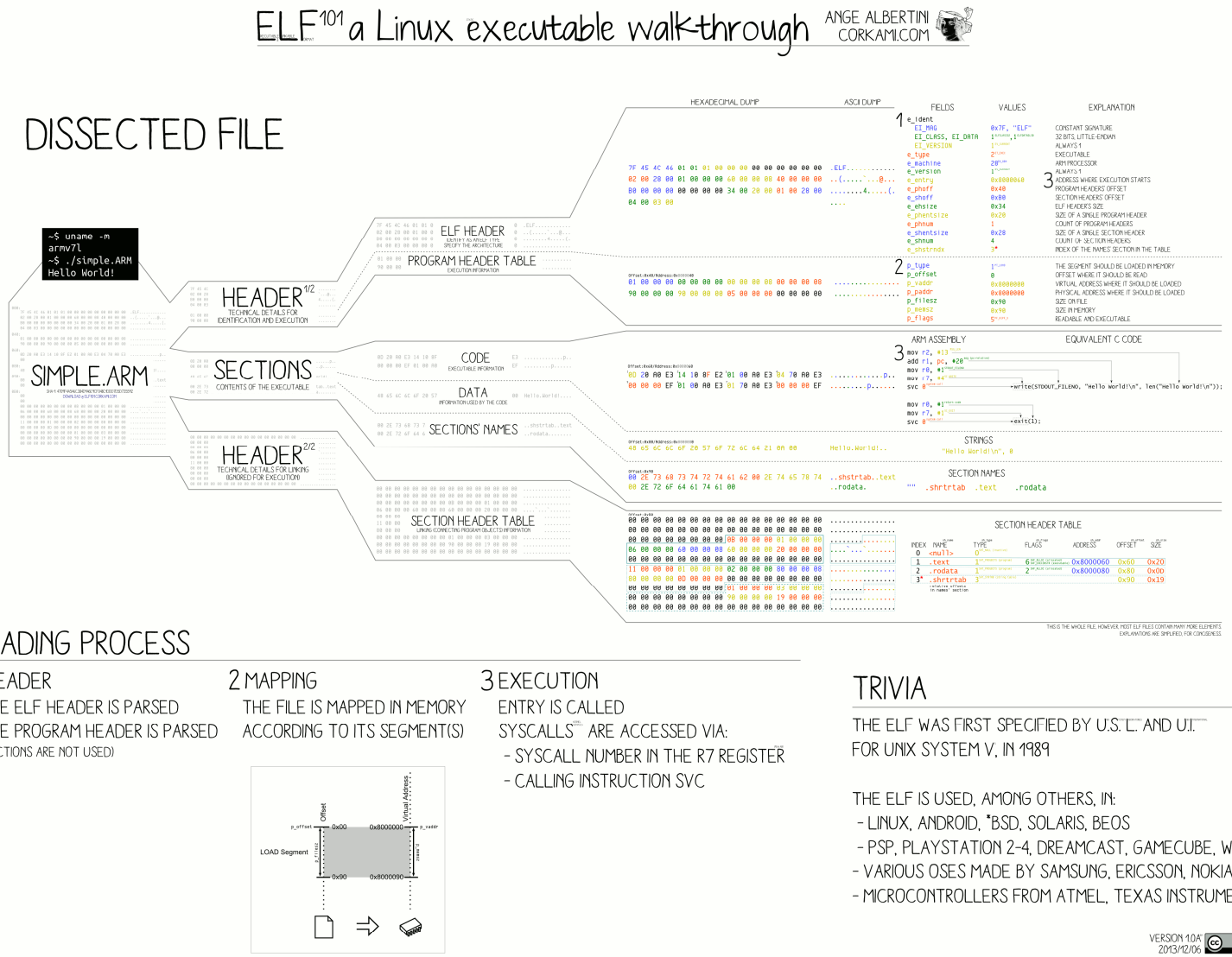


ELF file structure – 3

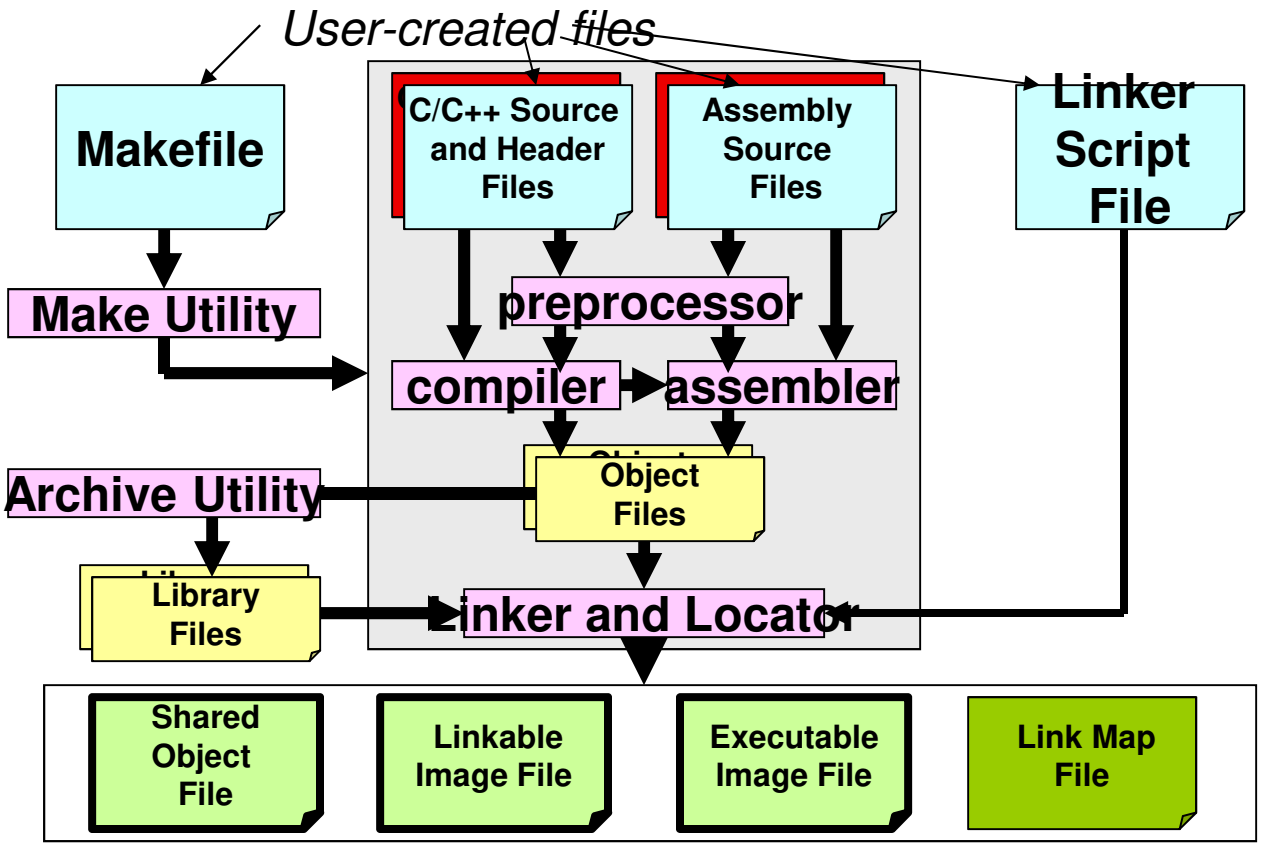
- **.symtab:** Symbol table.
- **.strtab:** Section for storing strings.
- **.init:** Process initialization code.
- **.fini:** Process termination code.
- **.debug:** Symbolic debug info.
- **.line:** Program source and machine code linking in debug.
- **.comment:** Extra info.



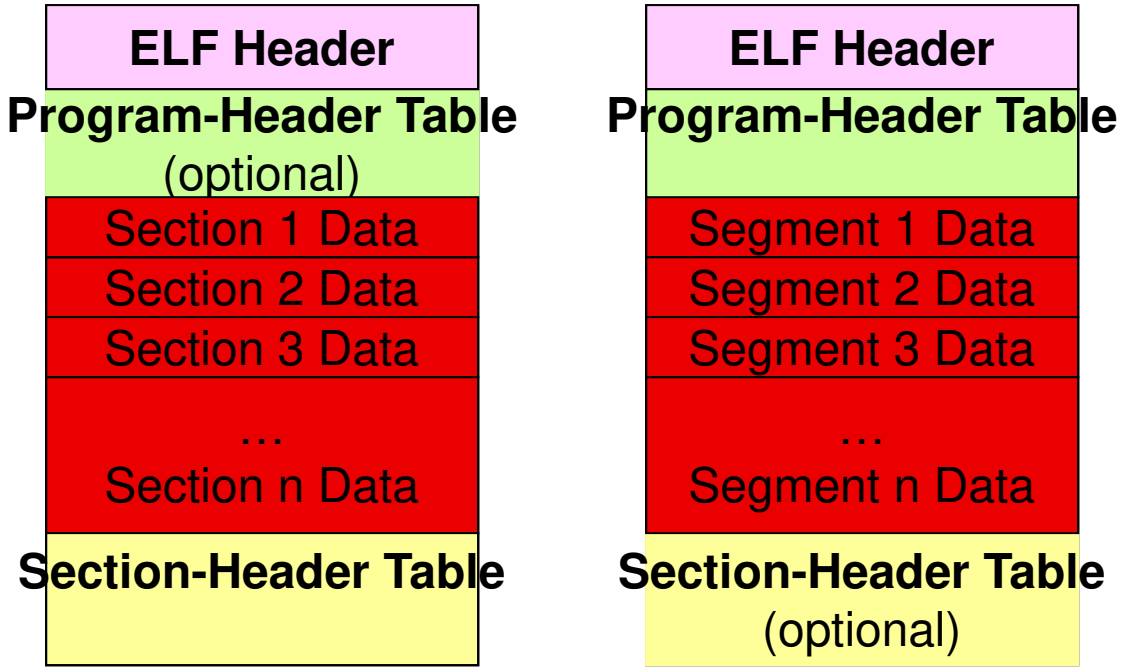
Executable and Linkable Format (ELF)



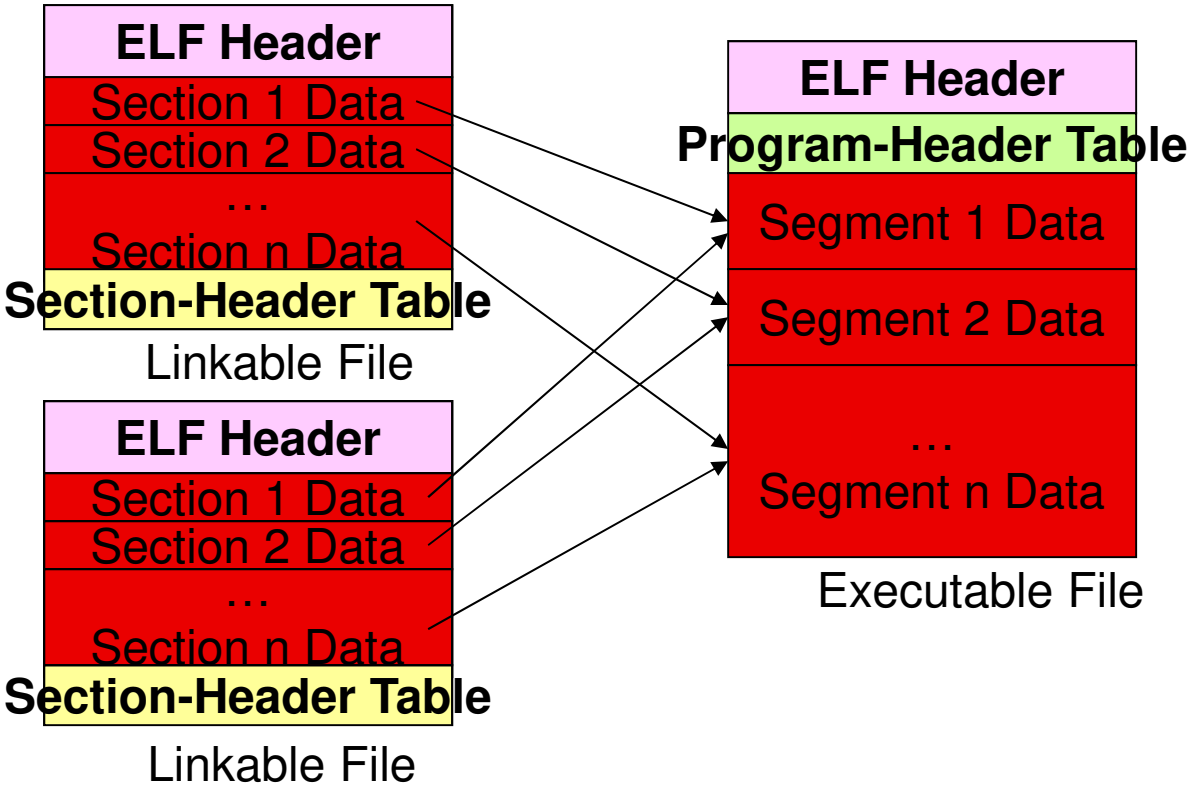
Overview of source translation



Executable versus Linkable



Role of the Linker



MALWARE STATIC ANALYSIS

Preparatory steps

- Read the first 3 chapters of the “Malware Data Science” book by Joshua Saxe & Hillary Sanders
- Download the book’s accompanying VirtualBox virtual machine from the website: https://www.malwaredatascience.com/ubuntu-virtual-machine
- Start the virtual machine

Sections in malware binaries

- Enter the ./malware_data_science/ch1 folder in the VM
- Save the following Python code to file “dissect.py”

```
import pefile
pe= pefile.PE("ircbot.exe")
for section in pe.sections:
    print(section.Name, hex(section.VirtualAddress),
hex(section.Misc_VirtualSize), section.SizeOfRawData)
```

- Inspect the PE file sections listed

DLLs loaded & functions called

- List the DLLs loaded and functions called from the malware file
  - Add to below code to “dissect.py” to get a hint about what the malware sample does once it is started

```
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    print entry.dll
    for function in entry.imports:
            print('\t', function.name)
```

- Investigate the list of DLL and their functions called

Inspecting the string resources

- Execute the following command line to export string resource to a TXT file:

```
strings ircbot.exe > ircbotstrings.txt
```

- Open the output in the “vim” editor:

```
vim ircbotstrings.txt
```

- Find lines indicating that the malware is an HTTP server

```
[HTTPD]: Error: server failed,
returned: <%d>.
GET
HTTP/1.0 200 OK
Server: myBot
Cache-Control: no-cache,no-
store,max-age=0
pragma: no-cache
Content-Type: %s
Content-Length: %i
Accept-Ranges: bytes
Date: %s %s GMT
Last-Modified: %s %s GMT
Expires: %s %s GMT
Connection: close
HTTP/1.0 200 OK
```

Inspecting the code

- Create & run “disassemble.py” with the below code

```
import pefile
from capstone import *

#load & find code
pe = pefile.PE("ircbot.exe")
entrypoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint
entrypoint_address = entrypoint + pe.OPTIONAL_HEADER.ImageBase
binary_code = pe.get_memory_mapped_image()[entrypoint:entrypoint+100]

# disassemble the code
disassembler = Cs(CS_ARCH_X86, CS_MODE_32)
for instruction in disassembler.disasm(binary_code, entrypoint_address):
    print("%s\t$s", instruction.mnemonic, instruction.op_str)
```

Static analysis challenges

- **Packing:** Malware authors compress, encrypt or otherwise modify their malicious program → malware samples become inscrutable (i.e. not readable) to malware analysts.
  - Solution: Dynamic analysis, i.e. run the sample in a sandbox.
- **Resource obfuscation:** Malware authors obfuscate the way program resources (e.g. strings & graphical) images are stored on disk and then de-obfuscate them at runtime.
  - Solution: Dynamic analysis or manual de-obfuscation based on code analysis.
- **Anti-disassembly techniques:** Hide code or masquerade instructions
- **Dynamically downloaded data:** The malware sample loads its core malicious components from a remote server.

MALWARE DYNAMIC ANALYSIS

Typical malware behavior

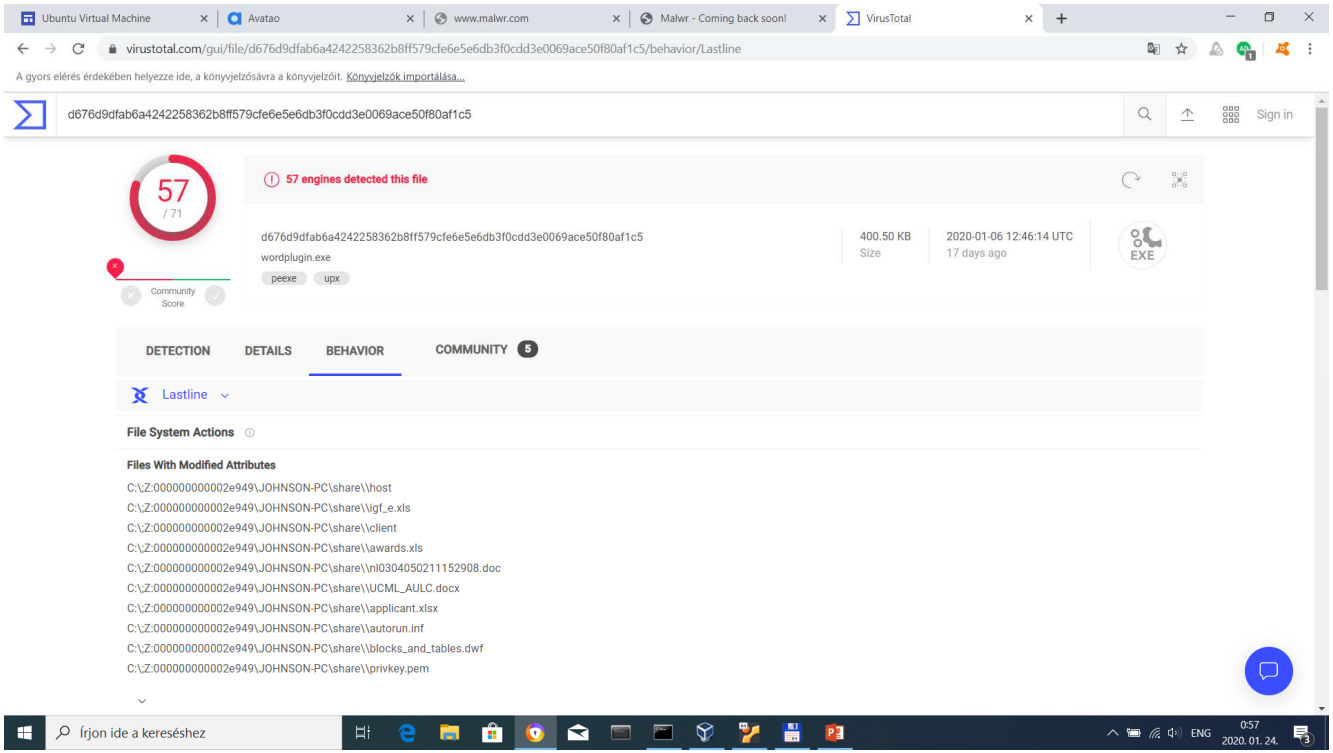
- **File system:** Write a device driver to disk, change system configuration files, add new programs to the file system and modify registry keys to ensure the malware auto-starts.
- **Windows registry:** Allows the malware to disable firewall or perform other similar unwanted actions.
- **Device drivers:** Load a driver which logs keyboard events.
- **Network actions:** Resolve domain names & make HTTP requests.

Investigate sample on VirusTotal.com

- Upload this malware sample file hash on VirusTotal: d676d9dfab6a4242258362b8ff579cfe6e5e6db3f0cdd3e0069ace50f80af1c5

- Inspect the dynamic analysis results with default Cuckoofork sandbox
- NOTE: Malwr.com is not available in January 2020

VirusTotal.com – Lastline results



VirusTotal.com behavior (Jan 2020)

Lastline sandbox

- File system actions
  - Modified file attributes
- Registry actions (set & delete)
- Process & service actions
  - Processes created
  - Shell commands
- Synchronization mechanisms
- Modules loaded, i.e. DLLs

Tencent HABO sandbox

- Behavior tags
- File system actions
  - Open, written, deleted, copied
- Registry actions
- Process & service actions
- Synchronization Mechanisms & Signals
- Modules Loaded
- Highlighted Actions, e.g. crypt algos observed

REVERSING CUSTOM PROGRAMS

Preparatory steps

- Access Avatao in a web browser (www.avatao.com)
- Log in
- Search for “Reverse”
- Open path “Reverse Engineering”
- Work through the 1st challenge entitled “Assembly Basics Tutorial”
- Work through the 2nd challenge entitled “GDB Basics Tutorial”

ADDITIONAL CHALLENGES

Avatao

- The “Lab 4: Reverse engineering Android Applications” path
- Elements of the “Hacktivity 2015” path
- Elements of the “Bootstrapping IT Security 2016” path

Summary

- Portable Executable (PE)
- Executable and Linkable Format (ELF)
- Malware analysis
  - Static
  - Dynamic
- Reversing custom programs

Thank you for your attention!