Maria Stancu 300243486

Assignment 3 – SRE reversing

Contents

Environment setup 2

Problem 1 4

A. ELF Sections 4

File header 4

Program header 5

Section headers 6

B. Object file 7

C. Describe ELF sections 9

.text 9

.data 9

.rodata 10

.bss 10

Problem 2 13

A. Propose a more optimized way to crack the program. 13

B. Possible optimization is to make our “strcmp” always succeed. 14

C. What could go wrong with a “strcmp” that always succeeds 15

Problem 3 Hardening against interposing: 16

Problem 4 Crackme challenge 16

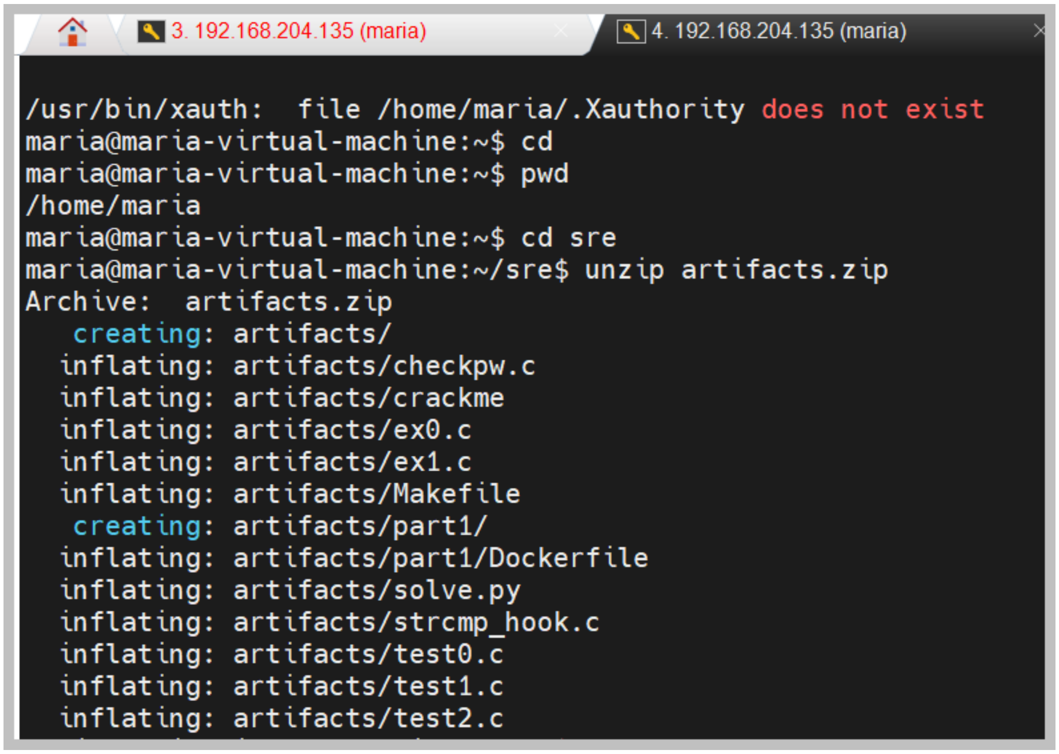
Solution 1 16

Solution 2 16

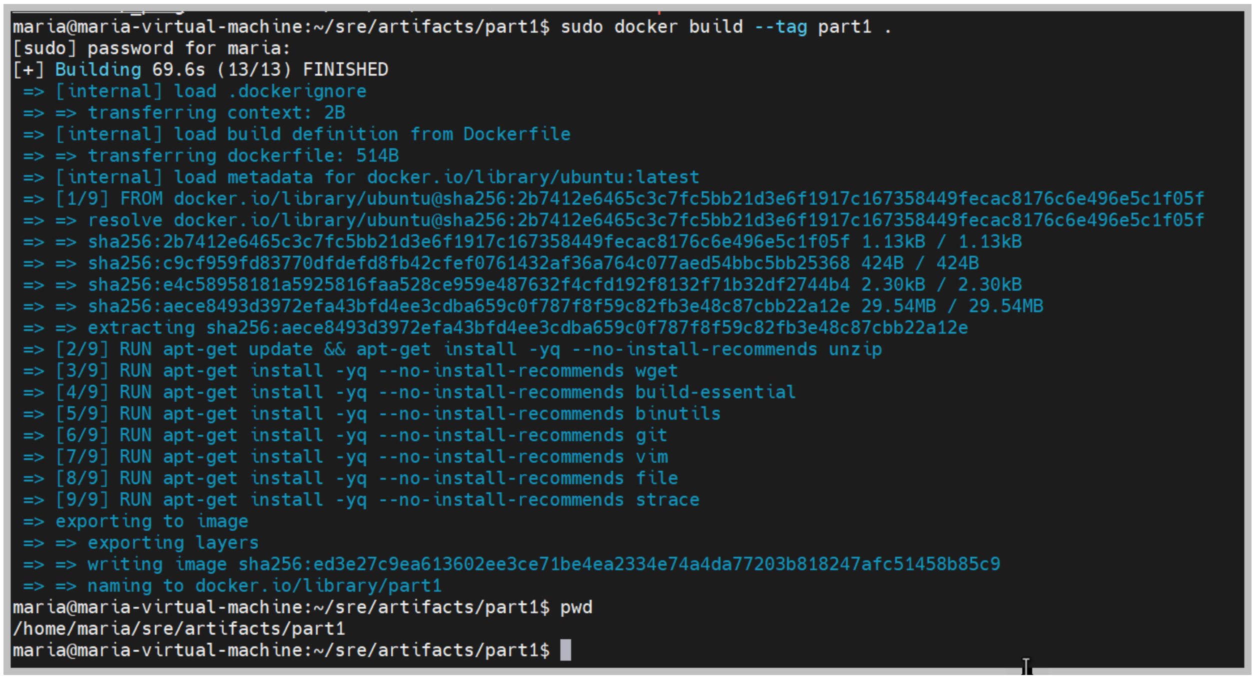
Solution 3 17

References: 18

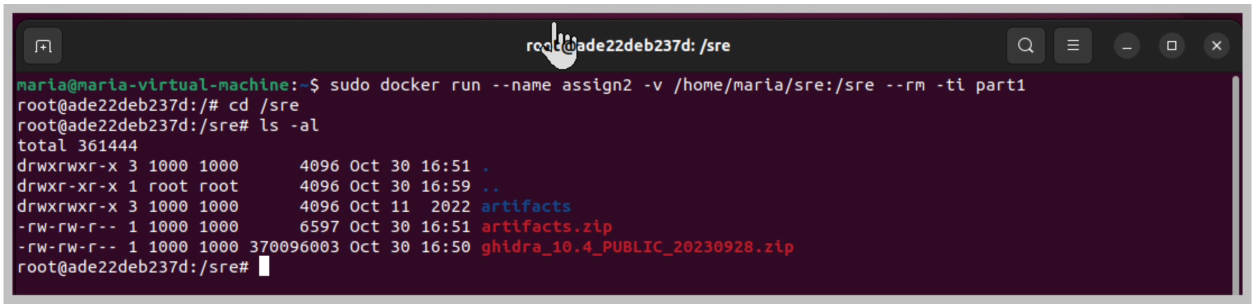
Environment setup



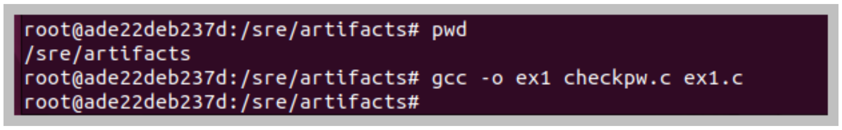
```
/usr/bin/xauth:  file /home/maria/.Xauthority does not exist
maria@maria-virtual-machine:~$ cd
maria@maria-virtual-machine:~$ pwd
/home/maria
maria@maria-virtual-machine:~$ cd sre
maria@maria-virtual-machine:~/sre$ unzip artifacts.zip
Archive:  artifacts.zip
   creating: artifacts/
  inflating: artifacts/checkpw.c
  inflating: artifacts/crackme
  inflating: artifacts/ex0.c
  inflating: artifacts/ex1.c
  inflating: artifacts/Makefile
   creating: artifacts/part1/
  inflating: artifacts/part1/Dockerfile
  inflating: artifacts/solve.py
  inflating: artifacts/strcmp_hook.c
  inflating: artifacts/test0.c
  inflating: artifacts/test1.c
  inflating: artifacts/test2.c
```



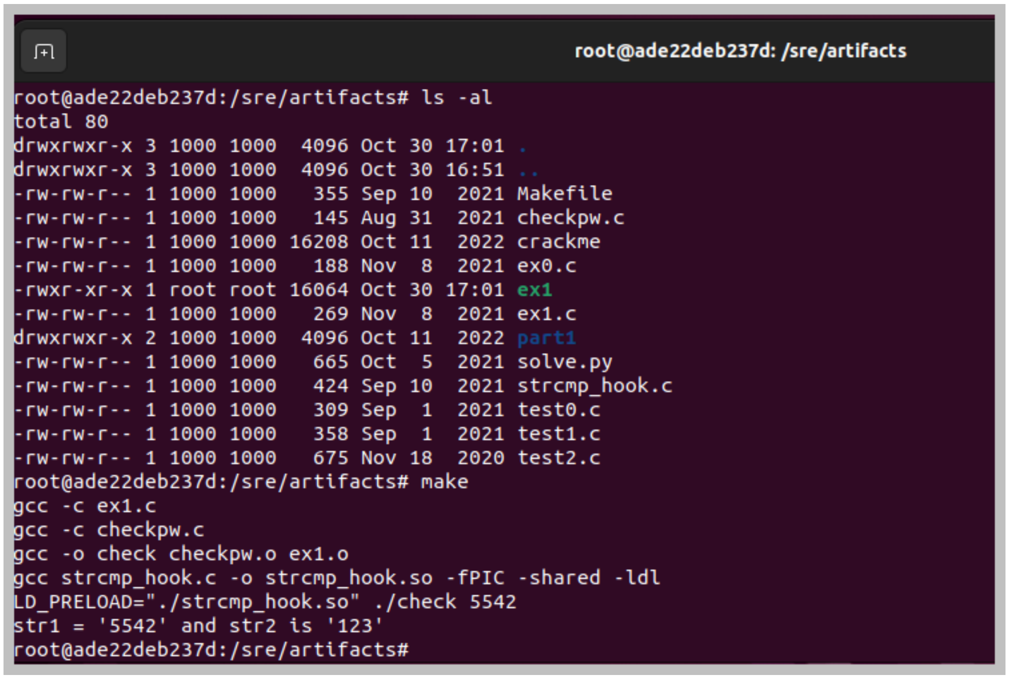
```
maria@maria-virtual-machine:~/sre/artifacts/part1$ sudo docker build --tag part1 .
[sudo] password for maria:
[+] Building 69.6s (13/13) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 514B
 => [internal] load metadata for docker.io/library/ubuntu:latest
 => [1/9] FROM docker.io/library/ubuntu@sha256:2b7412e6465c3c7fc5bb21d3e6f1917c167358449fecac8176c6e496e5c1f05f
 => => resolve docker.io/library/ubuntu@sha256:2b7412e6465c3c7fc5bb21d3e6f1917c167358449fecac8176c6e496e5c1f05f
 => => sha256:2b7412e6465c3c7fc5bb21d3e6f1917c167358449fecac8176c6e496e5c1f05f 1.13kB / 1.13kB
 => => sha256:c9cf959fd83770dfdefd8fb42cfef0761432af36a764c077aed54bbc5bb25368 424B / 424B
 => => sha256:e4c58958181a5925816faa528ce959e487632f4cfd192f8132f71b32df2744b4 2.30kB / 2.30kB
 => => sha256:aece8493d3972efa43bfd4ee3cdba659c0f787f8f59c82fb3e48c87cbb22a12e 29.54MB / 29.54MB
 => => extracting sha256:aece8493d3972efa43bfd4ee3cdba659c0f787f8f59c82fb3e48c87cbb22a12e
 => [2/9] RUN apt-get update && apt-get install -yq --no-install-recommends unzip
 => [3/9] RUN apt-get install -yq --no-install-recommends wget
 => [4/9] RUN apt-get install -yq --no-install-recommends build-essential
 => [5/9] RUN apt-get install -yq --no-install-recommends binutils
 => [6/9] RUN apt-get install -yq --no-install-recommends git
 => [7/9] RUN apt-get install -yq --no-install-recommends vim
 => [8/9] RUN apt-get install -yq --no-install-recommends file
 => [9/9] RUN apt-get install -yq --no-install-recommends strace
 => exporting to image
 => => exporting layers
 => => writing image sha256:ed3e27c9ea613602ee3ce71be4ea2334e74a4da77203b818247afc51458b85c9
 => => naming to docker.io/library/part1
maria@maria-virtual-machine:~/sre/artifacts/part1$ pwd
/home/maria/sre/artifacts/part1
maria@maria-virtual-machine:~/sre/artifacts/part1$ 
```



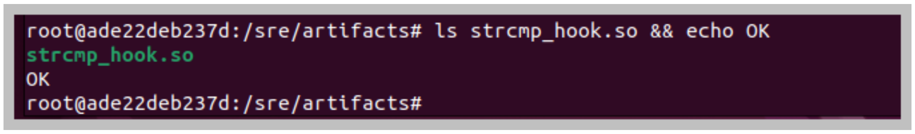
```
maria@maria-virtual-machine:~$ sudo docker run --name assign2 -v /home/maria/sre:/sre --rm -ti part1
root@ade22deb237d:/# cd /sre
root@ade22deb237d:/sre# ls -al
total 361444
drwxrwxr-x 3 1000 1000      4096 Oct 30 16:51 .
drwxr-xr-x 1 root root      4096 Oct 30 16:59 ..
drwxrwxr-x 3 1000 1000      4096 Oct 11  2022 artifacts
-rw-rw-r-- 1 1000 1000      6597 Oct 30 16:51 artifacts.zip
-rw-rw-r-- 1 1000 1000 370096003 Oct 30 16:50 ghidra_10.4_PUBLIC_20230928.zip
root@ade22deb237d:/sre# 
```



```
root@ade22deb237d:/sre/artifacts# pwd
/sre/artifacts
root@ade22deb237d:/sre/artifacts# gcc -o ex1 checkpw.c ex1.c
root@ade22deb237d:/sre/artifacts#
```



```
root@ade22deb237d:/sre/artifacts# ls -al
total 80
drwxrwxr-x 3 1000 1000  4096 Oct 30 17:01 .
drwxrwxr-x 3 1000 1000  4096 Oct 30 16:51 ..
-rw-rw-r-- 1 1000 1000   355 Sep 10  2021 Makefile
-rw-rw-r-- 1 1000 1000   145 Aug 31  2021 checkpw.c
-rw-rw-r-- 1 1000 1000 16208 Oct 11  2022 crackme
-rw-rw-r-- 1 1000 1000   188 Nov  8  2021 ex0.c
-rwxr-xr-x 1 root root 16064 Oct 30 17:01 ex1
-rw-rw-r-- 1 1000 1000   269 Nov  8  2021 ex1.c
drwxrwxr-x 2 1000 1000  4096 Oct 11  2022 part1
-rw-rw-r-- 1 1000 1000   665 Oct  5  2021 solve.py
-rw-rw-r-- 1 1000 1000   424 Sep 10  2021 strcmp_hook.c
-rw-rw-r-- 1 1000 1000   309 Sep  1  2021 test0.c
-rw-rw-r-- 1 1000 1000   358 Sep  1  2021 test1.c
-rw-rw-r-- 1 1000 1000   675 Nov 18  2020 test2.c
root@ade22deb237d:/sre/artifacts# make
gcc -c ex1.c
gcc -c checkpw.c
gcc -o check checkpw.o ex1.o
gcc strcmp_hook.c -o strcmp_hook.so -fPIC -shared -ldl
LD_PRELOAD="./strcmp_hook.so" ./check 5542
str1 = '5542' and str2 is '123'
root@ade22deb237d:/sre/artifacts#
```



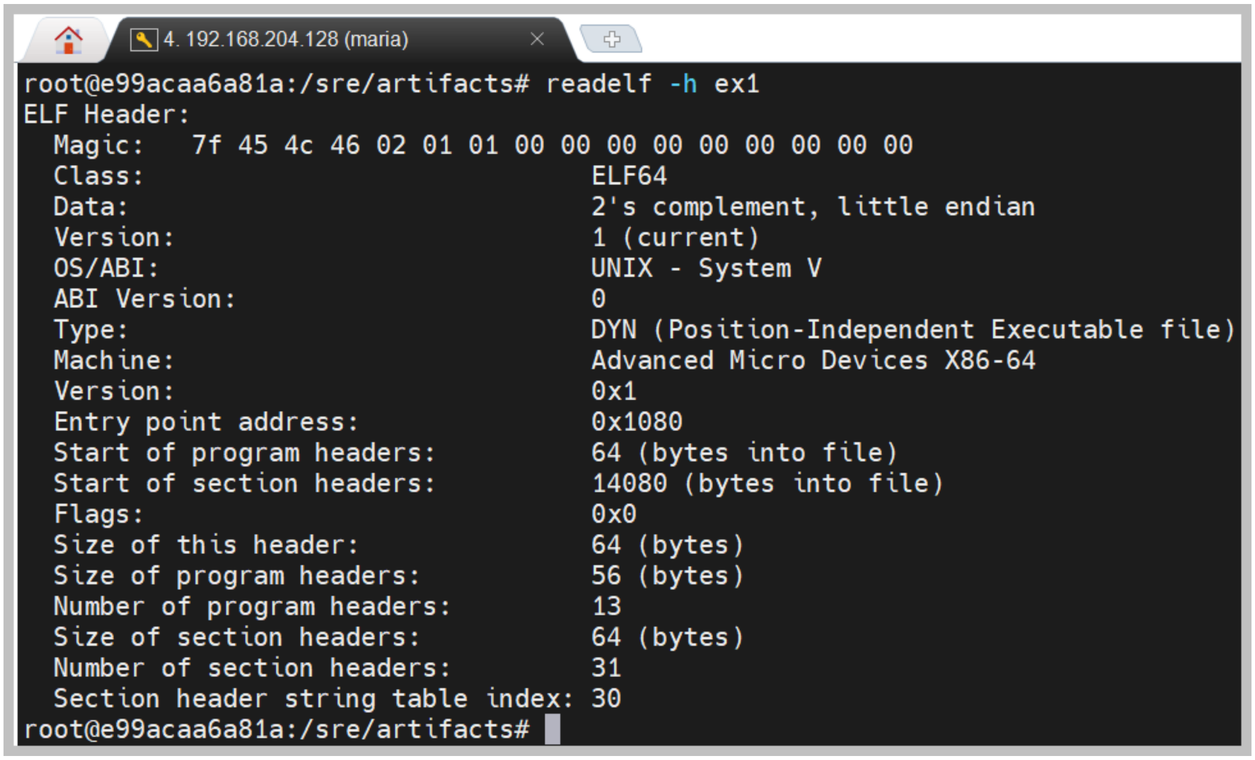
```
root@ade22deb237d:/sre/artifacts# ls strcmp_hook.so && echo OK
strcmp_hook.so
OK
root@ade22deb237d:/sre/artifacts#
```

Problem 1

A. ELF Sections

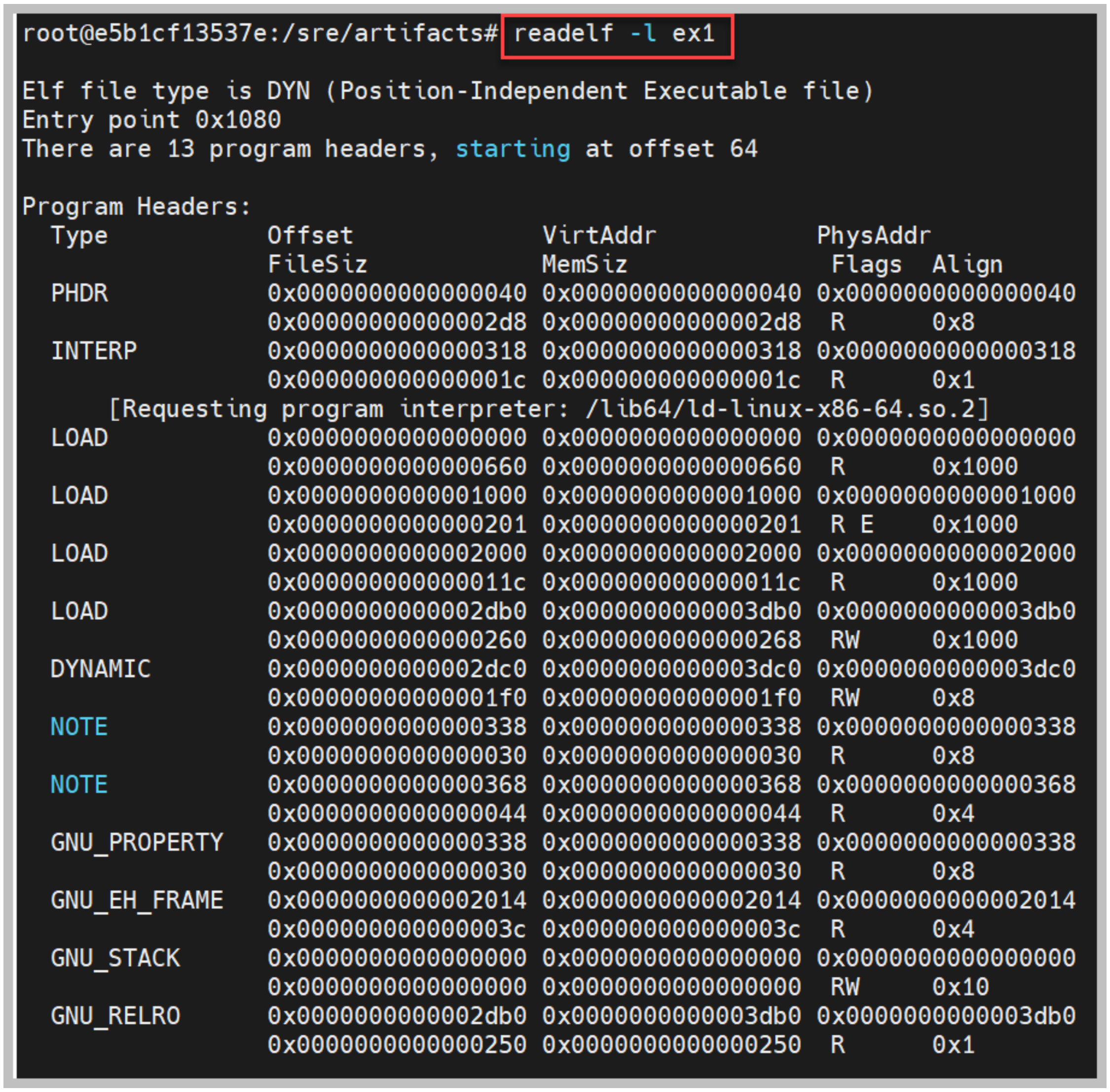
Show all the ELF sections in the ‘ex1’ executable (this could simply be the output of a tool that you run on the executable).

File header



```
root@e99acaa6a81a:/sre/artifacts# readelf -h ex1
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:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14080 (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
root@e99acaa6a81a:/sre/artifacts#
```

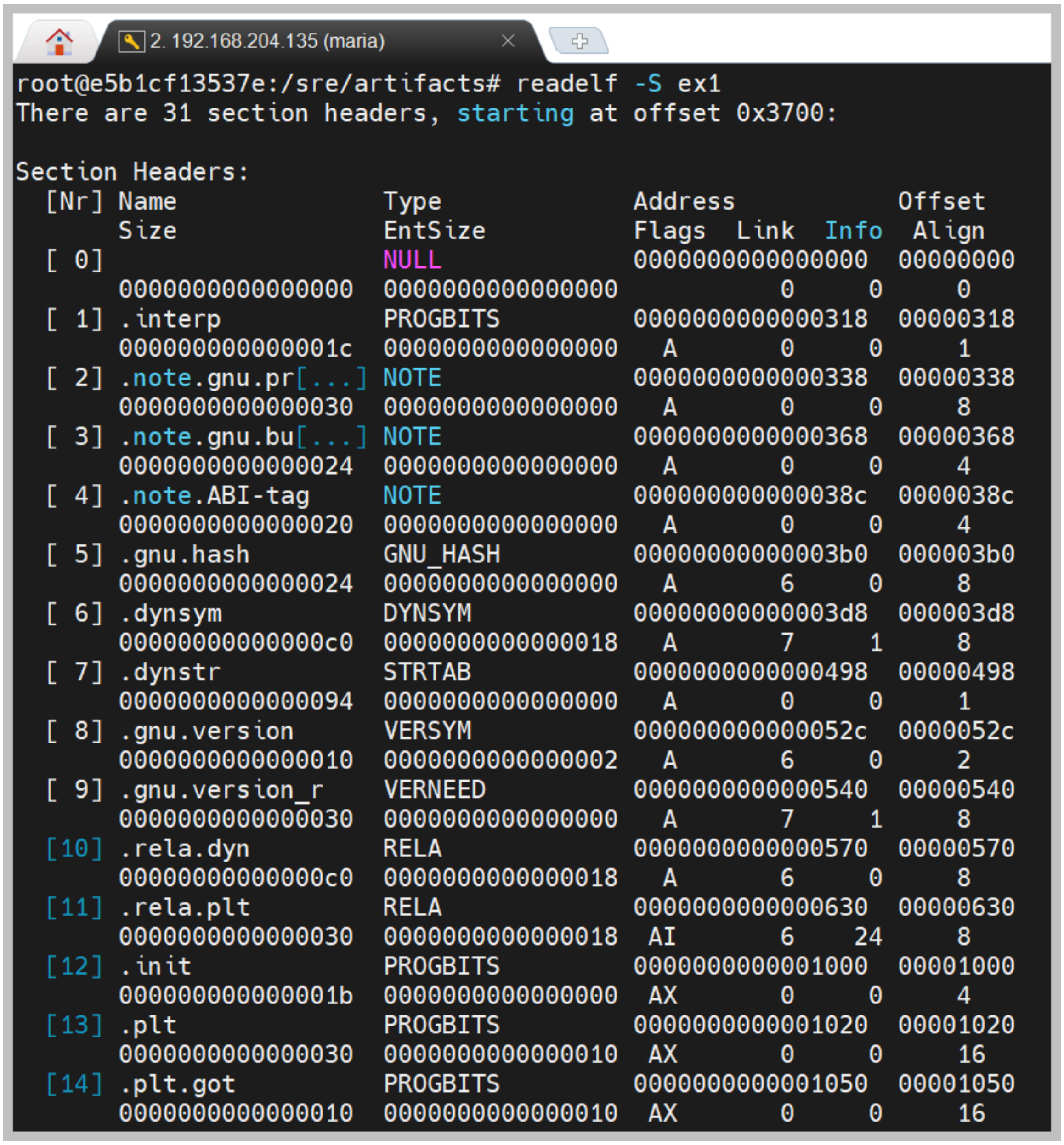
Program header



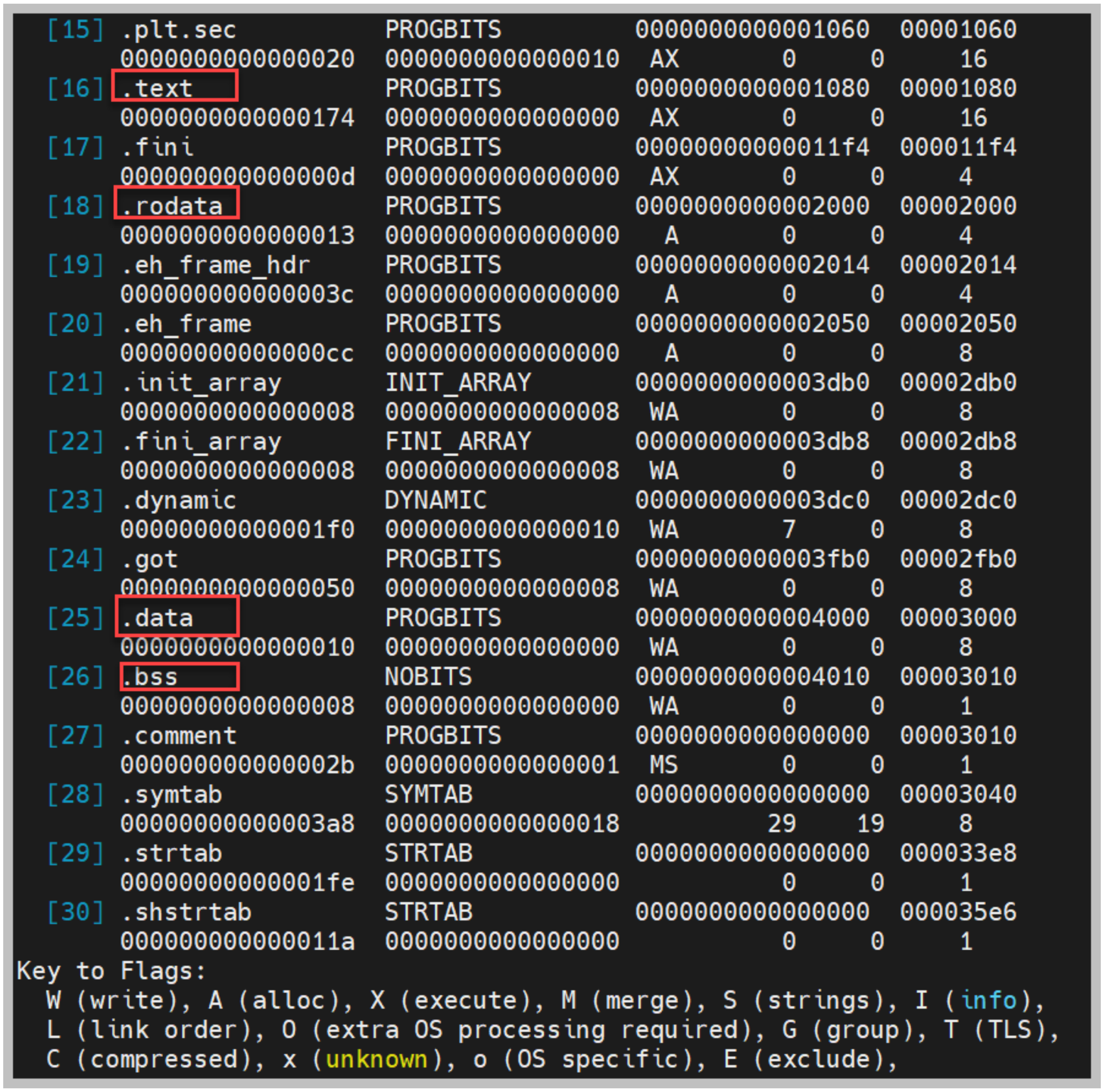
```
root@e5b1cf13537e:/sre/artifacts# readelf -l ex1

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000660 0x0000000000000660  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000201 0x0000000000000201  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x000000000000011c 0x000000000000011c  R      0x1000
  LOAD           0x0000000000002db0 0x0000000000003db0 0x0000000000003db0
                 0x0000000000000260 0x0000000000000268  RW     0x1000
  DYNAMIC        0x0000000000002dc0 0x0000000000003dc0 0x0000000000003dc0
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002014 0x0000000000002014 0x0000000000002014
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db0 0x0000000000003db0 0x0000000000003db0
                 0x0000000000000250 0x0000000000000250  R      0x1
```

Section headers

For executable files there are four main sections: **.text**, **.data**, **.rodata**, and **.bss**. Each of these sections is loaded with different access rights.



```
root@e5b1cf13537e:/sre/artifacts# readelf -S ex1
There are 31 section headers, starting at offset 0x3700:

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
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       00000000000000c0  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000498  00000498
       0000000000000094  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           000000000000052c  0000052c
       0000000000000010  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000540  00000540
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000570  00000570
       00000000000000c0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000000630  00000630
       0000000000000030  0000000000000018  AI       6    24     8
  [12] .init             PROGBITS         0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000001020  00001020
       0000000000000030  0000000000000010  AX       0     0     16
  [14] .plt.got          PROGBITS         0000000000001050  00001050
       0000000000000010  0000000000000010  AX       0     0     16
```



```
  [15] .plt.sec          PROGBITS         0000000000001060  00001060
       0000000000000020  0000000000000010  AX       0     0     16
  [16] .text             PROGBITS         0000000000001080  00001080
       0000000000000174  0000000000000000  AX       0     0     16
  [17] .fini             PROGBITS         00000000000011f4  000011f4
       000000000000000d  0000000000000000  AX       0     0     4
  [18] .rodata           PROGBITS         0000000000002000  00002000
       0000000000000013  0000000000000000   A       0     0     4
  [19] .eh_frame_hdr     PROGBITS         0000000000002014  00002014
       000000000000003c  0000000000000000   A       0     0     4
  [20] .eh_frame         PROGBITS         0000000000002050  00002050
       00000000000000cc  0000000000000000   A       0     0     8
  [21] .init_array       INIT_ARRAY       0000000000003db0  00002db0
       0000000000000008  0000000000000008  WA       0     0     8
  [22] .fini_array       FINI_ARRAY       0000000000003db8  00002db8
       0000000000000008  0000000000000008  WA       0     0     8
  [23] .dynamic          DYNAMIC          0000000000003dc0  00002dc0
       00000000000001f0  0000000000000010  WA       7     0     8
  [24] .got              PROGBITS         0000000000003fb0  00002fb0
       0000000000000050  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000004000  00003000
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000004010  00003010
       0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00003010
       000000000000002b  0000000000000001  MS       0     0     1
  [28] .symtab           SYMTAB           0000000000000000  00003040
       00000000000003a8  0000000000000018          29    19     8
  [29] .strtab           STRTAB           0000000000000000  000033e8
       00000000000001fe  0000000000000000           0     0     1
  [30] .shstrtab         STRTAB           0000000000000000  000035e6
       000000000000011a  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
```

B. Object file

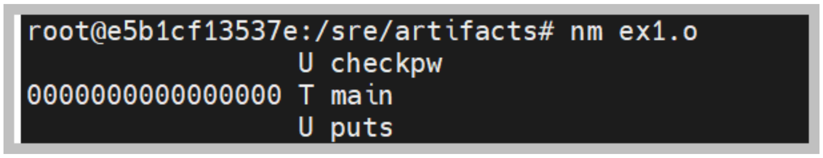
What is the difference between an object file (e.g. ‘gcc -c ex1.c will produce ex1.o’) file and an executable (e.g. ‘gcc -o ex1 checkpw.c ex1.c will produce ex1’)? (Hint: you can talk about the symbol ‘main’ and it’s properties in each file type).

To create an application from a C source file, the source files are first compiled into object code and then the object files created by the compiler are linked to create the executable file.

Object files are complied into binary machine language and they contain unresolved external references. They may need to be linked against other object files, C/C++ runtime library or third party libraries.

Object files contain machine code and also contains metadata about the addresses of its variables and functions (called symbols).

Symbols (functions and variables) can be displayed for an object file (function main is listed as a symbol).



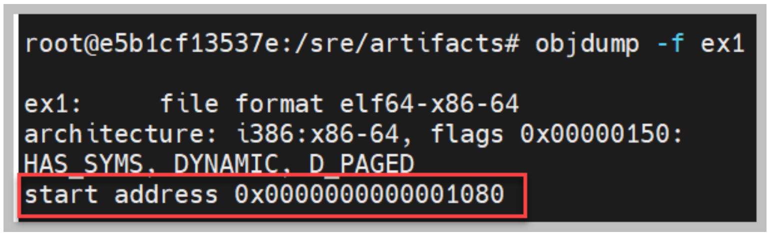
```
root@e5b1cf13537e:/sre/artifacts# nm ex1.o
                 U checkpw
0000000000000000 T main
                 U puts
```

Object files reference each other using symbols: if program A calls a function “functionB()” which resides in program B, then program A will contain a symbol “functionB()” and a location where the address is stored. Program B will have a symbol “functionB()” with its address.

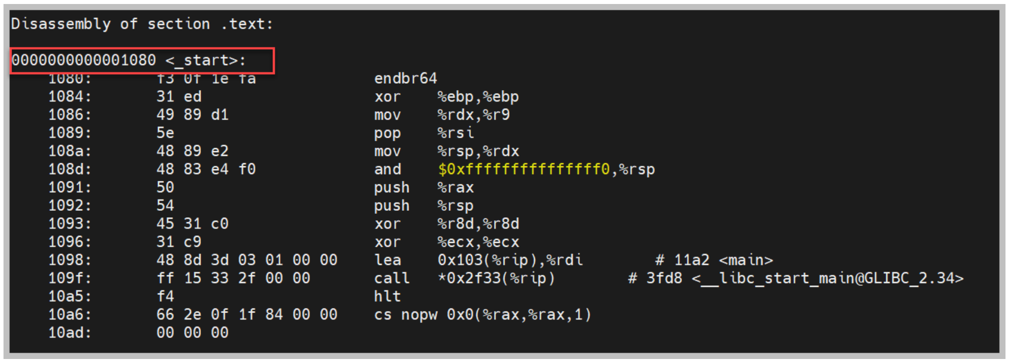
The executable file created after compiling the C source code is an Executable and Linkable Format file.

Every ELF file has an ELF header where there is a **e_entry** field which contains the program memory address from which the execution of executable will start. This memory address point to the _**start()** function.

For ex1:



```
root@e5b1cf13537e:/sre/artifacts# objdump -f ex1

ex1:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000001080
```



```
Disassembly of section .text:

0000000000001080 <_start>:
    1080:       f3 0f 1e fa             endbr64
    1084:       31 ed                   xor    %ebp,%ebp
    1086:       49 89 d1                mov    %rdx,%r9
    1089:       5e                      pop    %rsi
    108a:       48 89 e2                mov    %rsp,%rdx
    108d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1091:       50                      push   %rax
    1092:       54                      push   %rsp
    1093:       45 31 c0                xor    %r8d,%r8d
    1096:       31 c9                   xor    %ecx,%ecx
    1098:       48 8d 3d 03 01 00 00    lea    0x103(%rip),%rdi        # 11a2 <main>
    109f:       ff 15 33 2f 00 00       call   *0x2f33(%rip)        # 3fd8 <__libc_start_main@GLIBC_2.34>
    10a5:       f4                      hlt
    10a6:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    10ad:       00 00 00
```

The _start() function prepare the input arguments for another function **_libc_start_main()** which will be called next.

After all the prerequisite actions has been completed, _libc_start_main() calls the main() function.

Main() – from the object file – is the agreed function for startup code. We could use any other function as the startup point. The _start() calls by default the main() function – in case we want to execute any custom code we will need to change the _start() function

C. Describe ELF sections

Describe what these sections are: text, data, rodata and bss. Add a small C program ("elfsections.c") where you show the difference between "not initialized" and "initialized" data in the resulting elf executable (Hint: read up on the sections in ELF).

.text

This section holds the instructions that the program needs for it to run. Contains executable code. It will be packed into a segment with read and execute access rights. It is only loaded once, as the contents will not change. This can be seen with the **objdump** utility.

.data

Initialized data, with read/write access rights. The data segment is read/write, since the values of variables can be altered at run time.

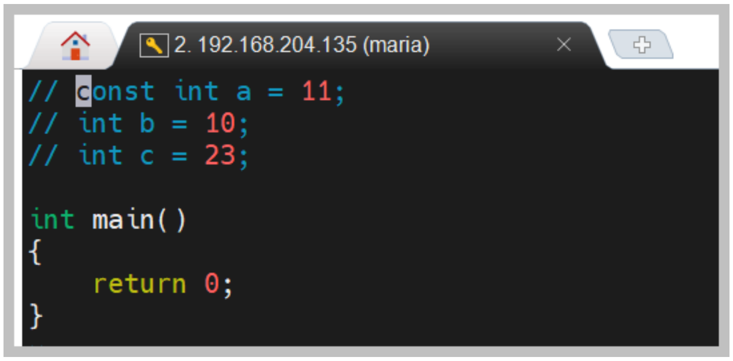
.rodata

Initialized data, with read access rights only (=A).

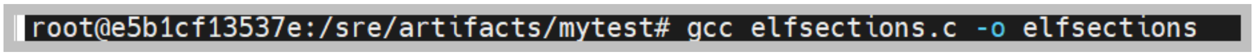
.bss

Uninitialized data, with read/write access rights (=WA). Variables and constants.

Created a simple C program:

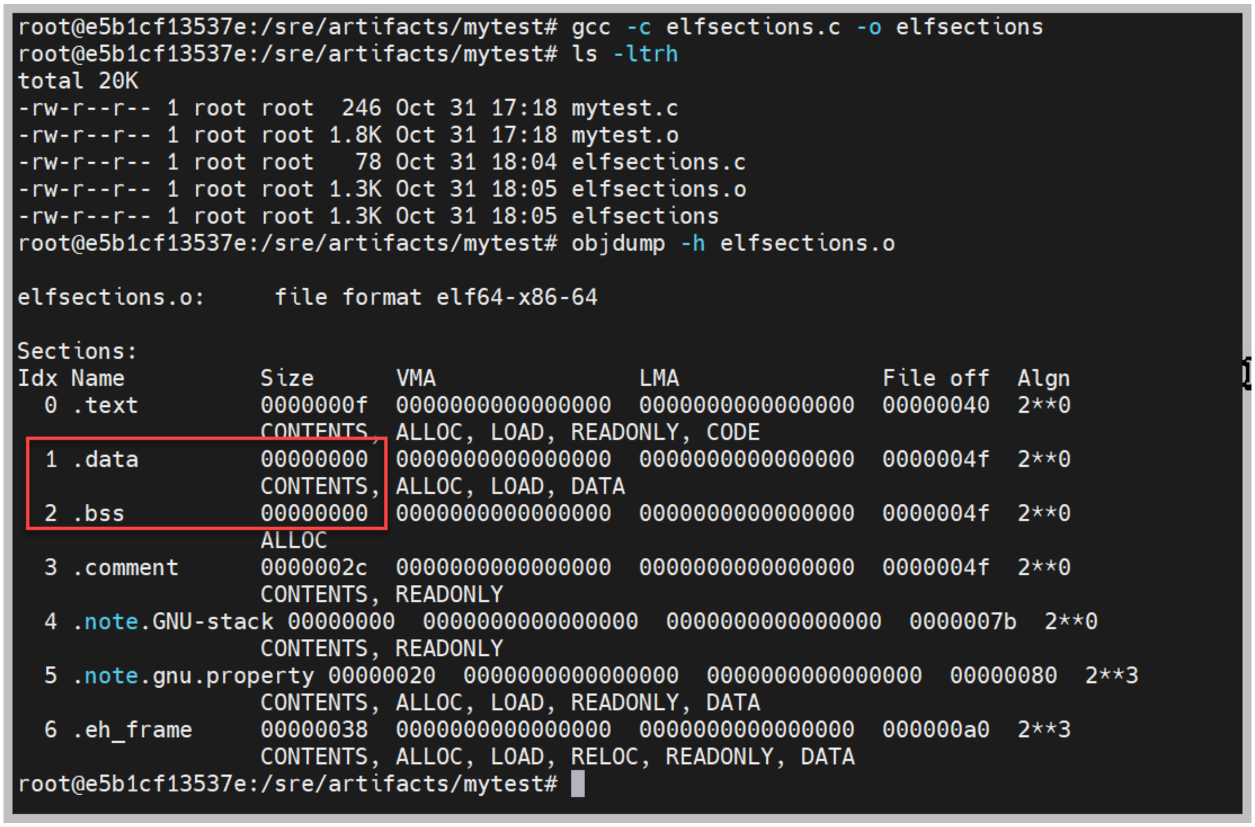


```
// const int a = 11;
// int b = 10;
// int c = 23;

int main()
{
        return 0;
}
```



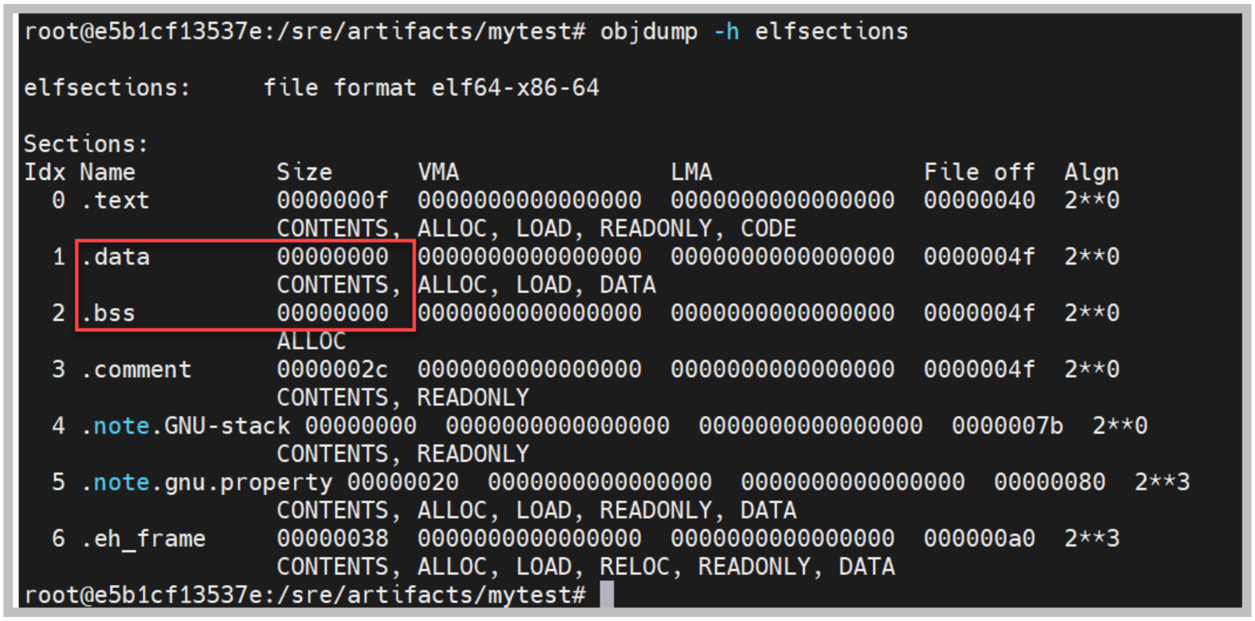
```
root@e5b1cf13537e:/sre/artifacts/mytest# gcc elfsections.c -o elfsections
```

➔ Display the sections and sizes of the object file:



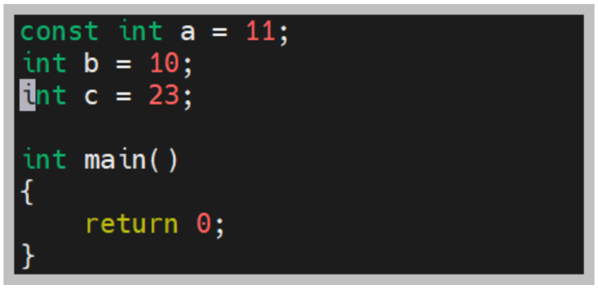
```
root@e5b1cf13537e:/sre/artifacts/mytest# gcc -c elfsections.c -o elfsections
root@e5b1cf13537e:/sre/artifacts/mytest# ls -ltrh
total 20K
-rw-r--r-- 1 root root  246 Oct 31 17:18 mytest.c
-rw-r--r-- 1 root root 1.8K Oct 31 17:18 mytest.o
-rw-r--r-- 1 root root   78 Oct 31 18:04 elfsections.c
-rw-r--r-- 1 root root 1.3K Oct 31 18:05 elfsections.o
-rw-r--r-- 1 root root 1.3K Oct 31 18:05 elfsections
root@e5b1cf13537e:/sre/artifacts/mytest# objdump -h elfsections.o

elfsections.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000000f  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  0000004f  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  0000004f  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  0000004f  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000007b  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000080  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000038  0000000000000000  0000000000000000  000000a0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
root@e5b1cf13537e:/sre/artifacts/mytest# 
```

- ➔ Display the sections and sizes of executable:



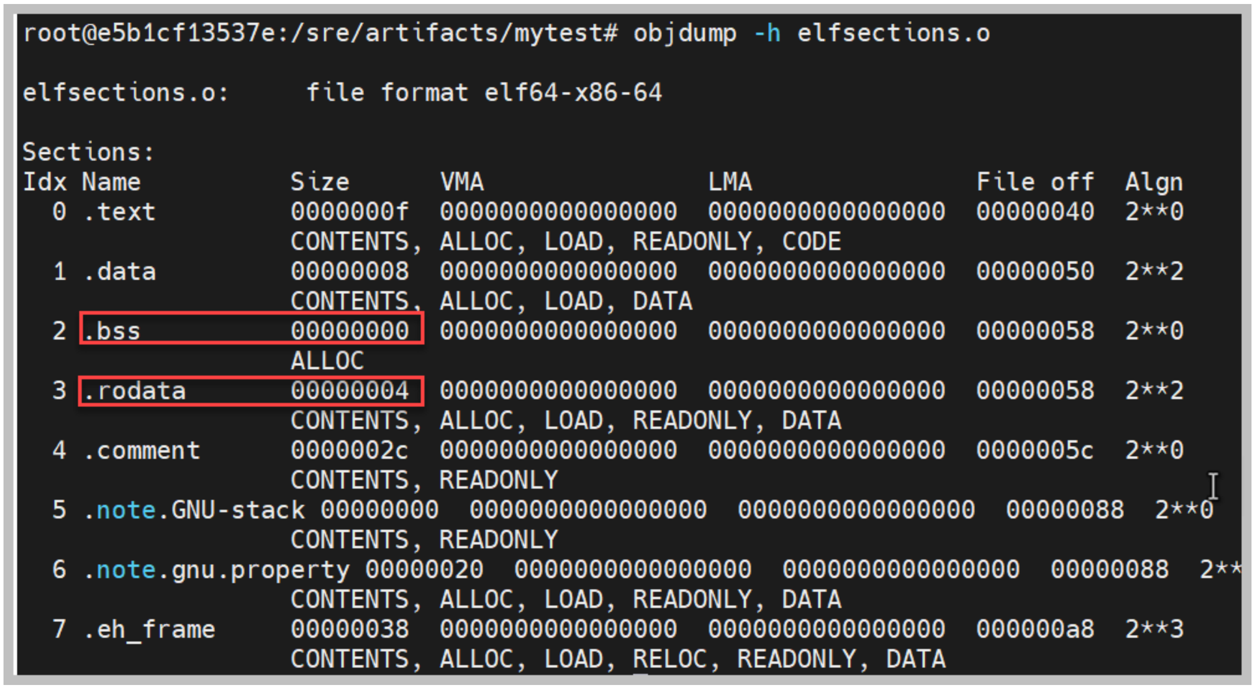
```
root@e5b1cf13537e:/sre/artifacts/mytest# objdump -h elfsections

elfsections:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000000f  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  0000004f  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  0000004f  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  0000004f  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000007b  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000080  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000038  0000000000000000  0000000000000000  000000a0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
root@e5b1cf13537e:/sre/artifacts/mytest#
```

- ➔ Uncomment the variables and recompile



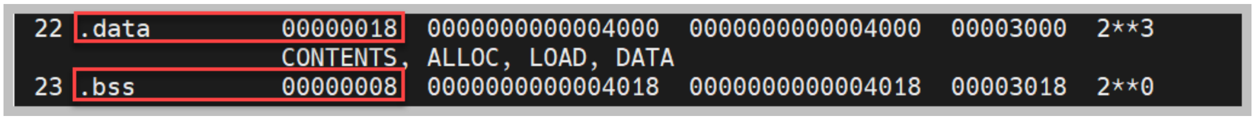
```
const int a = 11;
int b = 10;
int c = 23;

int main()
{
        return 0;
}
```

- ➔ gcc -c elfsections.c -o elfsections
- ➔ Display the sections and sizes of the object file:



```
root@e5b1cf13537e:/sre/artifacts/mytest# objdump -h elfsections.o

elfsections.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000000f  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000050  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000058  2**0
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000002c  0000000000000000  0000000000000000  0000005c  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  00000088  2**0
                  CONTENTS, READONLY
  6 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000088  2**
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .eh_frame     00000038  0000000000000000  0000000000000000  000000a8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

- ➔ Display the sections and sizes of executable:

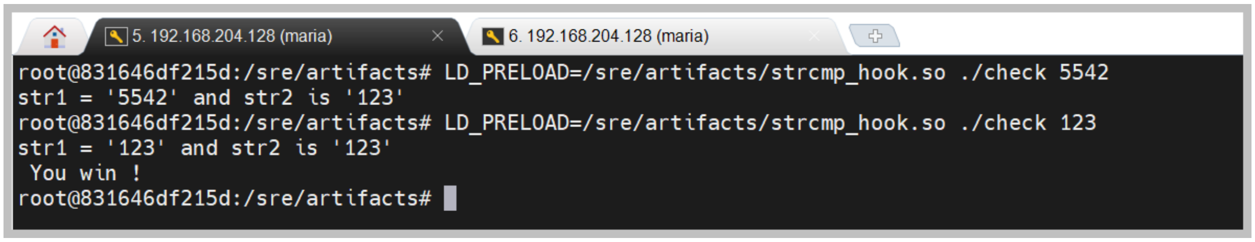


```
 22 .data         00000018  0000000000004000  0000000000004000  00003000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .bss          00000008  0000000000004018  0000000000004018  00003018  2**0
```

Problem 2

A. Propose a more optimized way to crack the program.

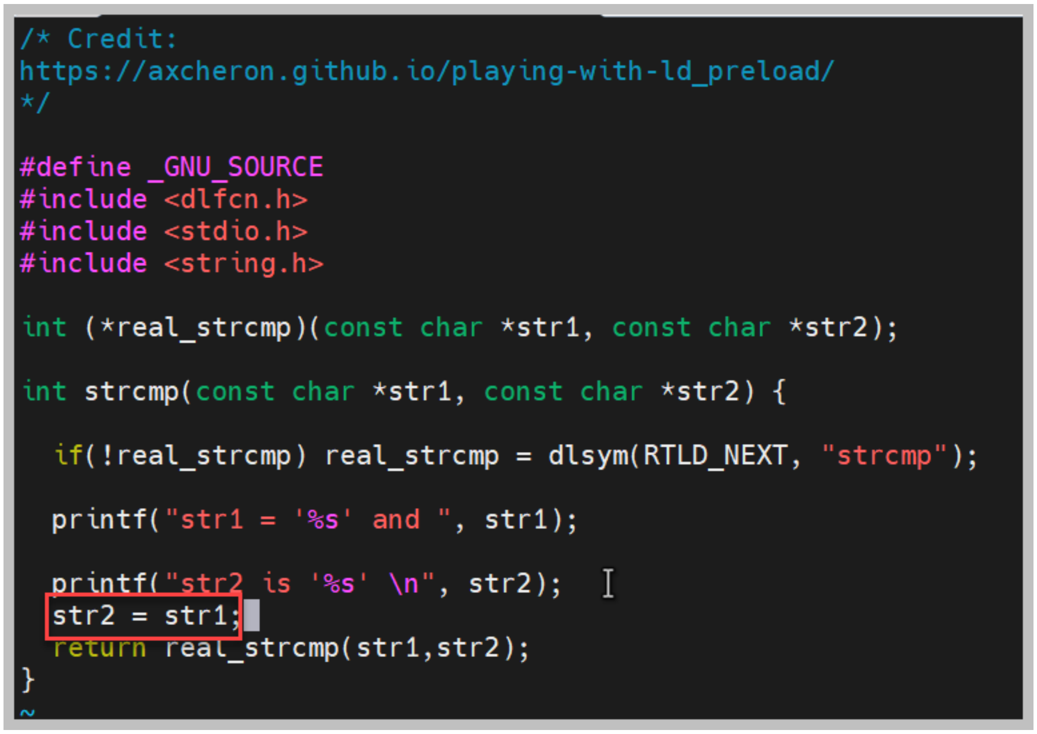
Test:



```
root@831646df215d:/sre/artifacts# LD_PRELOAD=/sre/artifacts/strcmp_hook.so ./check 5542
str1 = '5542' and str2 is '123'
root@831646df215d:/sre/artifacts# LD_PRELOAD=/sre/artifacts/strcmp_hook.so ./check 123
str1 = '123' and str2 is '123'
 You win !
root@831646df215d:/sre/artifacts# 
```

We could modify the strcmp_hook.c file in such a way that str2=str1, in this way, it doesn't matter which string you are typing, the program will always succeed because str2 will be equal to str1

- ➔ I created strcmp_hook_test.c file

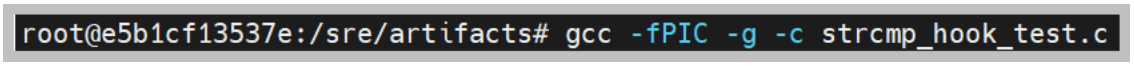


```
/* Credit:
https://axcheron.github.io/playing-with-ld_preload/
*/

#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

int (*real_strcmp)(const char *str1, const char *str2);

int strcmp(const char *str1, const char *str2) {

  if(!real_strcmp) real_strcmp = dlsym(RTLD_NEXT, "strcmp");

  printf("str1 = '%s' and ", str1);

  printf("str2 is '%s' \n", str2);
  str2 = str1;
  return real_strcmp(str1,str2);
}
~
```

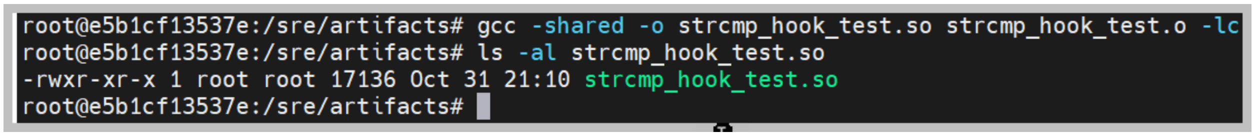
- ➔ Used lab5 for the compile command in order to create an .so file

gcc -fPIC -g -c strcmp_hook_test.c – create the object file

gcc -shared -o strcmp_hook_test strcmp_hook_test.o -lc

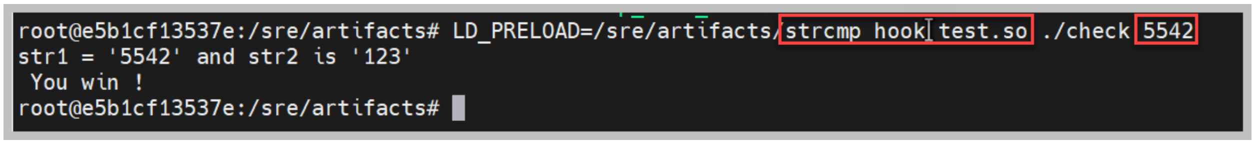


```
root@e5b1cf13537e:/sre/artifacts# gcc -fPIC -g -c strcmp_hook_test.c
```



```
root@e5b1cf13537e:/sre/artifacts# gcc -shared -o strcmp_hook_test.so strcmp_hook_test.o -lc
root@e5b1cf13537e:/sre/artifacts# ls -al strcmp_hook_test.so
-rwxr-xr-x 1 root root 17136 Oct 31 21:10 strcmp_hook_test.so
root@e5b1cf13537e:/sre/artifacts#
```

- ➔ Use my test library:



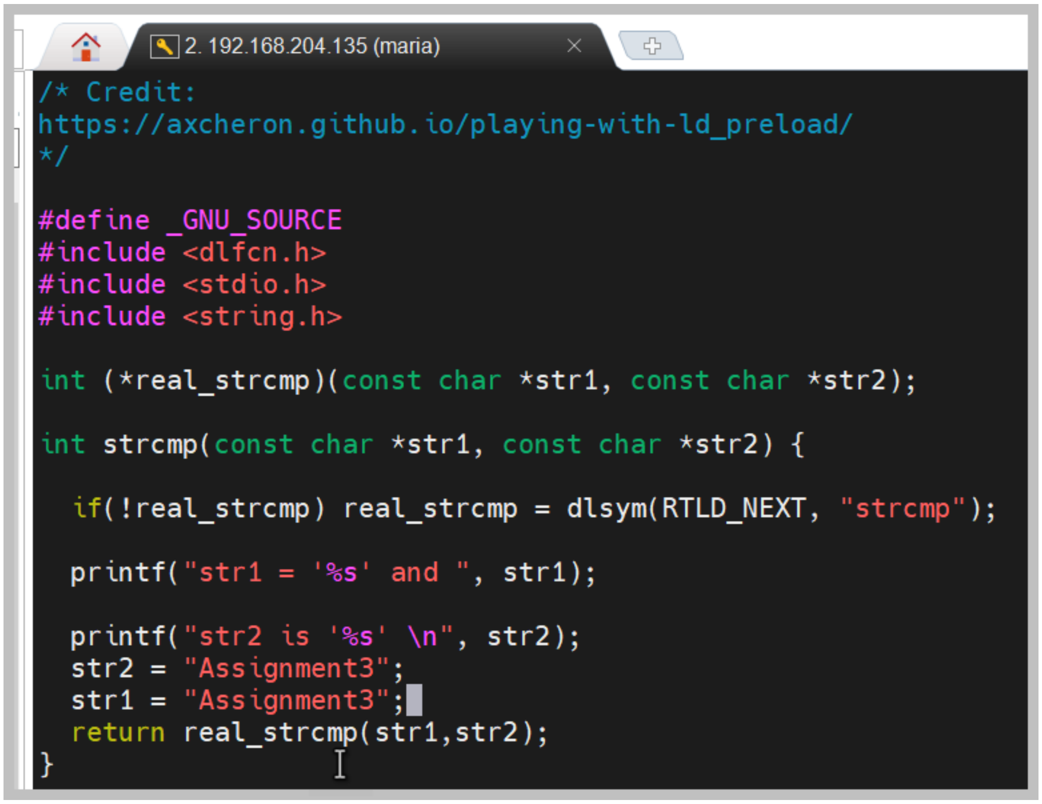
```
root@e5b1cf13537e:/sre/artifacts# LD_PRELOAD=/sre/artifacts/strcmp_hook_test.so ./check 5542
str1 = '5542' and str2 is '123'
 You win !
root@e5b1cf13537e:/sre/artifacts#
```

- ➔ By making str2=str1 in the shared library we always crack in the program irrespective of what we type (in the example above I typed 5542)

B. Possible optimization is to make our “strcmp” always succeed.

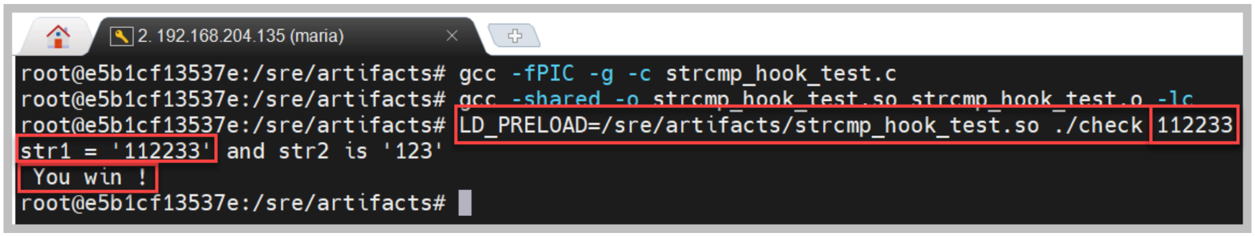
Another possibility to make strcmp always succeed would be to initialize str1 and str2 to the same value so that the call to the (real) strcmp function will always return true.

- ➔ Test using the custom strcmp_hook_test.c file



```
/* Credit:
https://axcheron.github.io/playing-with-ld_preload/
*/

#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

int (*real_strcmp)(const char *str1, const char *str2);

int strcmp(const char *str1, const char *str2) {

  if(!real_strcmp) real_strcmp = dlsym(RTLD_NEXT, "strcmp");

  printf("str1 = '%s' and ", str1);

  printf("str2 is '%s' \n", str2);
  str2 = "Assignment3";
  str1 = "Assignment3";
  return real_strcmp(str1,str2);
}
```

➔ Recompile and execute



```
root@e5b1cf13537e:/sre/artifacts# gcc -fPIC -g -c strcmp_hook_test.c
root@e5b1cf13537e:/sre/artifacts# gcc -shared -o strcmp_hook_test.so strcmp_hook_test.o -lc
root@e5b1cf13537e:/sre/artifacts# LD_PRELOAD=/sre/artifacts/strcmp_hook_test.so ./check 112233
str1 = '112233' and str2 is '123'
 You win !
root@e5b1cf13537e:/sre/artifacts#
```

C. What could go wrong with a “strcmp” that always succeeds

In case we have a program which always succeeds, that the branches in the conditional statements (if, while, do) are affected as well. The program will always gets executed only for the branches having <condition=True>.

Problem 3 Hardening against interposing:

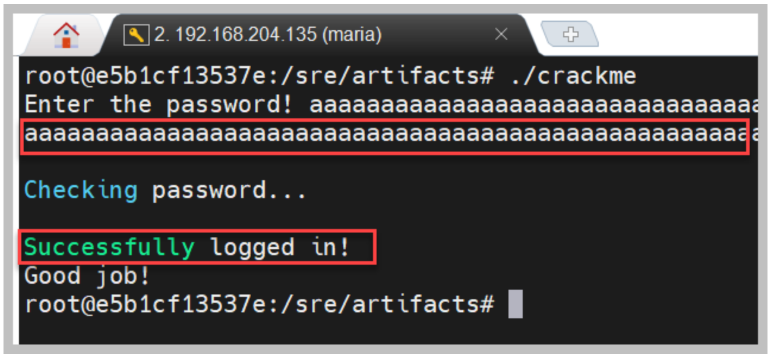
We know from Lab 5 that the LD* environment variables have an effect on the behavior of the shared libraries. LD_LIBRARY_PATH contains the path where the libraries should be searched first and LD_PRELOAD contains a list of custom shared libraries which should be loaded first. The LD* environment variables are not shared to the child process. By using an appropriate SETUID mechanism we can prevent the hijacking of the shared libraries.

- ➔ Also, we could unset the LD_PRELOAD environment variable before the compilation

Problem 4 Crackme challenge

Solution 1

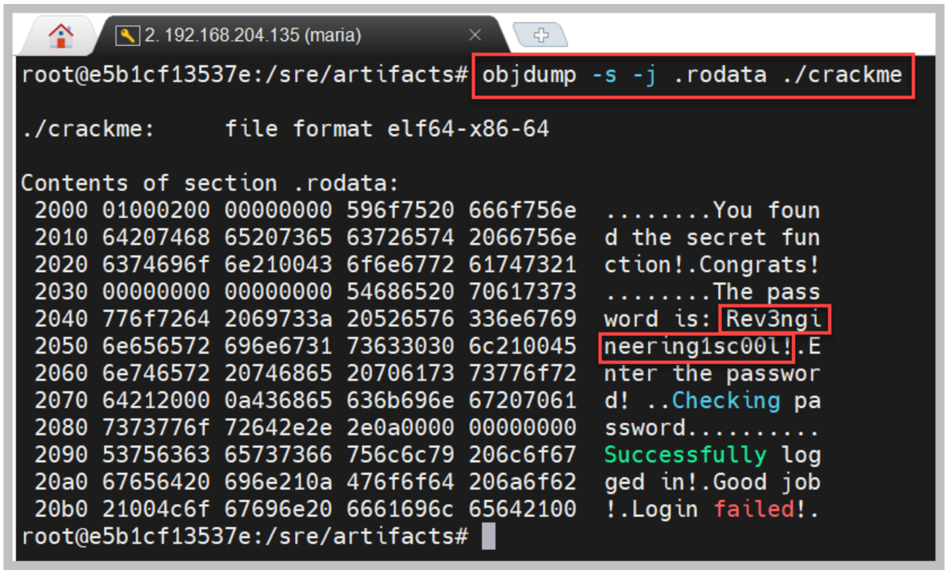
A buffer is an area of memory where data to be processed is stored. We are overflowing the buffer to change the value of the data. We overflow the buffer by specifying a long string containing the same characters.



```
root@e5b1cf13537e:/sre/artifacts# ./crackme
Enter the password! aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Checking password...

Successfully logged in!
Good job!
root@e5b1cf13537e:/sre/artifacts#
```

Solution 2

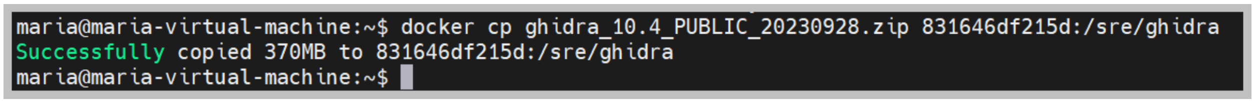
Exploiting the vulnerability in the crackme program, we could display the content of the .rodata section which contains the read-only data (texts)



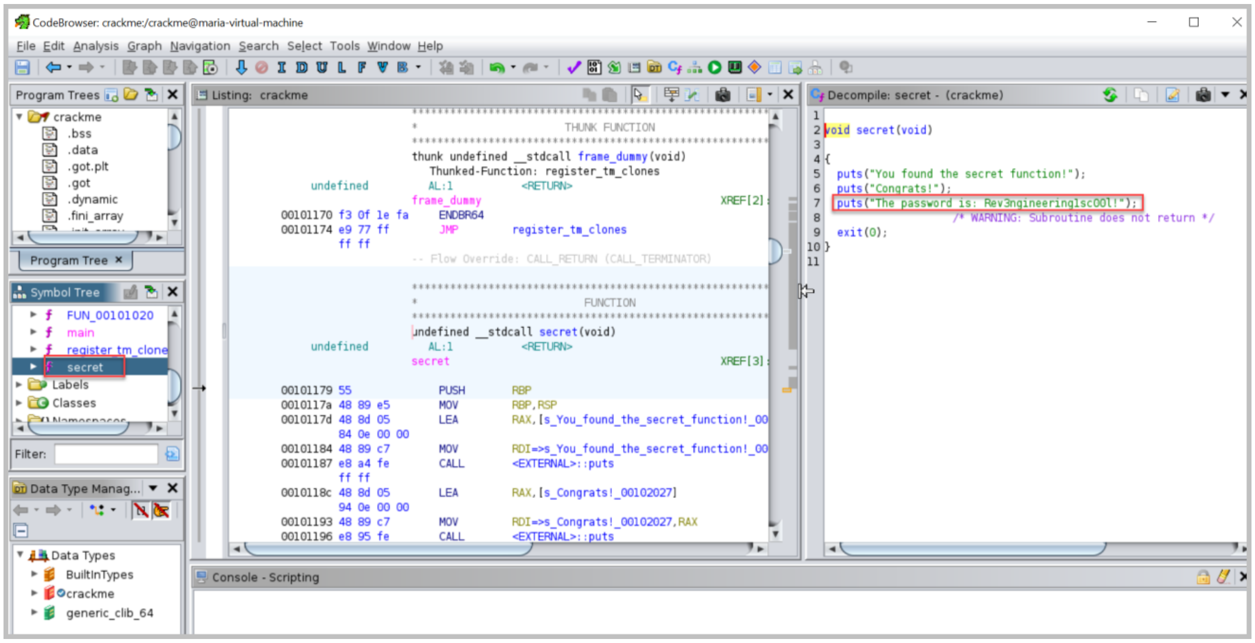
```
root@e5b1cf13537e:/sre/artifacts# objdump -s -j .rodata ./crackme

./crackme:     file format elf64-x86-64

Contents of section .rodata:
 2000 01000200 00000000 596f7520 666f756e  ........You foun
 2010 64207468 65207365 63726574 2066756e  d the secret fun
 2020 6374696f 6e210043 6f6e6772 61747321  ction!.Congrats!
 2030 00000000 00000000 54686520 70617373  ........The pass
 2040 776f7264 2069733a 20526576 336e6769  word is: Rev3ngi
 2050 6e656572 696e6731 73633030 6c210045  neering1sc00l!.E
 2060 6e746572 20746865 20706173 73776f72  nter the passwor
 2070 64212000 0a436865 636b696e 67207061  d! ..Checking pa
 2080 7373776f 72642e2e 2e0a0000 00000000  ssword..........
 2090 53756363 65737366 756c6c79 206c6f67  Successfully log
 20a0 67656420 696e210a 476f6f64 206a6f62  ged in!.Good job
 20b0 21004c6f 67696e20 6661696c 65642100  !.Login failed!.
root@e5b1cf13537e:/sre/artifacts# 
```

Solution 3

We are using the Ghidra program to display different sections of the program

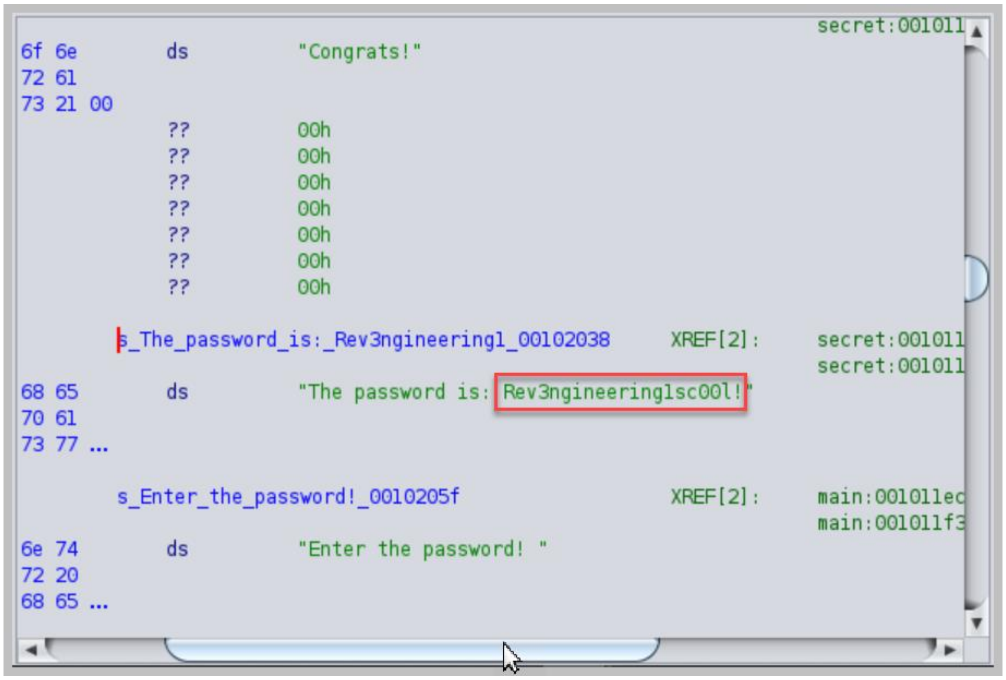


```
maria@maria-virtual-machine:~$ docker cp ghidra_10.4_PUBLIC_20230928.zip 831646df215d:/sre/ghidra
Successfully copied 370MB to 831646df215d:/sre/ghidra
maria@maria-virtual-machine:~$ 
```



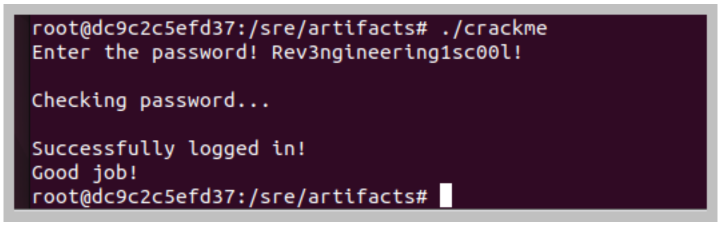
```
                         *                     THUNK FUNCTION
                         thunk undefined __stdcall frame_dummy(void)
                           Thunked-Function: register_tm_clones
         undefined         AL:1           <RETURN>
                         frame_dummy                                    XREF[2]
00101170 f3 0f 1e fa        ENDBR64
00101174 e9 77 ff           JMP          register_tm_clones
         ff ff
                         -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                         *                         FUNCTION
                         undefined __stdcall secret(void)
         undefined         AL:1           <RETURN>
                         secret                                         XREF[3]
00101179 55                 PUSH         RBP
0010117a 48 89 e5           MOV          RBP,RSP
0010117d 48 8d 05           LEA          RAX,[s_You_found_the_secret_function!_00
         84 0e 00 00
00101184 48 89 c7           MOV          RDI=>s_You_found_the_secret_function!_00
00101187 e8 a4 fe           CALL         <EXTERNAL>::puts
         ff ff
0010118c 48 8d 05           LEA          RAX,[s_Congrats!_00102027]
         94 0e 00 00
00101193 48 89 c7           MOV          RDI=>s_Congrats!_00102027,RAX
00101196 e8 95 fe           CALL         <EXTERNAL>::puts
```

```

void secret(void)

{
  puts("You found the secret function!");
  puts("Congrats!");
  puts("The password is: Rev3ngineering1sc00l!");
                    /* WARNING: Subroutine does not return */
  exit(0);
}

```



```
                                                              secret:001011
6f 6e        ds          "Congrats!"
72 61
73 21 00
             ??          00h
             ??          00h
             ??          00h
             ??          00h
             ??          00h
             ??          00h
             ??          00h

        s_The_password_is:_Rev3ngineering1_00102038      XREF[2]:     secret:001011
                                                                      secret:001011
68 65        ds          "The password is: Rev3ngineering1sc00l!"
70 61
73 77 ...

        s_Enter_the_password!_0010205f                   XREF[2]:     main:001011ec
                                                                      main:001011f3
6e 74        ds          "Enter the password! "
72 20
68 65 ...
```

➔ Test:



```
root@dc9c2c5efd37:/sre/artifacts# ./crackme
Enter the password! Rev3ngineering1sc00l!

Checking password...

Successfully logged in!
Good job!
root@dc9c2c5efd37:/sre/artifacts# 
```

References:

[What Is an ELF File? | Baeldung on Linux]

[The 101 of ELF files on Linux: Understanding and Analysis - Linux Audit (linux-audit.com)]

https://www.thegeekstuff.com/2012/09/objdump-examples/