cuckoo

Name: Cuckoo
Category: Exploiting
Description:

Hemos desarrollado un programa completamente resistente a cualquier tipo de ataque de desbordamiento de buffers ijaque mate jaquer!

Puedes descargar una réplica del mismo, o hablar con el original en la siguiente dirección:

nc 138.68.182.98:8001

Enumeration

Relevant information obtain by enumeration on the target file:

- The program stores its input in a buffer of 256 bytes which can be overflowed but, when overflowed, a method aborts the execution.
- There is a method **print_flag** declared in the code that is probably the key to obtain the flag.
- The name of the buffer is not contained in the symbol table.

First-Contact

The challenge provides a file ***cuckoo_hidden.elf***, the extension **.elf** means the file is an executable. The first step we are going to take is testing the exec. and try to find its vulnerabilities:

```
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ ./cuckoo_hidden.elf
a
Gracias por portarte bien :)
```

As we can see, while executing the code an input prompt is displayed. After inserting any character the message ***Gracias por portarte bien :)*** is returned. The description of the challenge states that the program is completely secure against buffer overflow, so we can try overflowing the input buffer:

```
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ ./cuckoo_hidden.elf
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Muy mal hecho :'(
```

As expected, after overflowing the buffer, the message ***Muy mal hecho :'(*** is returned. This means that the program actually is secure against buffer overflow. Let's build a python script to automatize the process of overflowing the buffer and try to scrap the size of the buffer:

```
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ python3 overload_script.py -n 255
-f cuckoo_hidden.elf
Gracias por portarte bien :)
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ python3 overload_script.py -n 256
-f cuckoo_hidden.elf
Muy mal hecho :'(
```

As expected, the buffer has a size of 256 bytes, meaning it can store 255 characters and the void **\0** character that denotes the end of a string.

At this point there is nothing more we can think about enumerating the execution of the program. So we run the progam strings ***cuckoo_hidden.elf*** and obtain the following output:

```
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ strings cuckoo_hidden.elf
4%> @
4%  @
La flag es flag{*************}
Gracias por portarte bien :)
Muy mal hecho :'(
the.asm
gracias
print_flag
read_line
read_loop
not_bad
sigue
len_start
print
__bss_start
_edata
_end
.symtab
.strtab
.shstrtab
.text
.data
```

The output shows many interesting paths to follow:

- The string **La flag es flag{*************}** which can be the result of the challenge, where the * characters can be overwritten with the real flag.
- The strings *Gracias por portarte bien :)* and *Muy mal hecho :'(* used to display messages to the user.
- The strings **gracias**, **print_flag**, **read_line**, **not_bad**, **sigue**, etc. which have a high probability of being function names in the code.

The next step is using gdb to debug the execution of the program:

```
(gdb) print (char*) &gracias
$10 = 0x402020 "Gracias por portarte bien :)\n"
(gdb) p (char*) &print_flag
$11 = 0x401000 <print_flag> "H\215\064%"
(gdb) p print_flag
$13 = {<text variable, no debug info>} 0x401000 <print_flag>
(gdb) call print_flag
$14 = {<text variable, no debug info>} 0x401000 <print_flag>
(gdb) disassemble print_flag
Dump of assembler code for function print_flag:
   0x0000000000401000 <+0>:     lea    0x402000,%rsi
   0x0000000000401008 <+8>:     callq  0x4010cb <print>
   0x000000000040100d <+13>:    retq
End of assembler dump.
(gdb) disassemble read_loop
Dump of assembler code for function read_loop:
   0x0000000000401031 <+0>:     mov    $0x0,%eax
   0x0000000000401036 <+5>:     mov    $0x0,%edi
   0x000000000040103b <+10>:    xor    %rdx,%rdx
   0x000000000040103e <+13>:    mov    -0x102(%rbp),%dx
   0x0000000000401045 <+20>:    lea    -0x100(%rbp,%rdx,1),%rsi
   0x000000000040104d <+28>:    xor    %rdx,%rdx
   0x0000000000401050 <+31>:    mov    $0x1,%edx
   0x0000000000401055 <+36>:    syscall
   0x0000000000401057 <+38>:    mov    (%rsi),%dl
   0x0000000000401059 <+40>:    xor    %cx,%cx
   0x000000000040105c <+43>:    mov    -0x102(%rbp),%cx
   0x0000000000401063 <+50>:    inc    %cx
   0x0000000000401066 <+53>:    mov    %cx,-0x102(%rbp)
   0x000000000040106d <+60>:    cmp    $0xa,%dl
   0x0000000000401070 <+63>:    jne    0x401031 <read_loop>
   0x0000000000401072 <+65>:    xor    %dx,%dx
   0x0000000000401075 <+68>:    mov    -0x102(%rbp),%dx
   0x000000000040107c <+75>:    movb   $0x0,-0x103(%rbp,%rdx,1)
   0x0000000000401084 <+83>:    add    $0x102,%rsp
   0x000000000040108b <+90>:    pop    %rdx
   0x000000000040108c <+91>:    cmp    $0xcafe,%edx
   0x0000000000401092 <+97>:    je     0x4010a3 <not_bad>
   0x0000000000401094 <+99>:    lea    0x40203e,%rsi
   0x000000000040109c <+107>:   callq  0x4010cb <print>
   0x00000000004010a1 <+112>:   jmp    0x4010b0 <sigue>
End of assembler dump.
```

As we can see in the code box, the execution of the program consists on an apparent loop which asks the user for input and ends when the user inserts a valid character or overflows the buffer. We are having trouble to find some of the strings and definitions discovered in the **strings** command, so we will use the utility **elftoc** from **elfkickers**' utilities:

```
alex@DESKTOP-MQKMDU5:/mnt/c/Ciberseg/cuckoo$ ../Utilities/elfkickers/elftoc
cuckoo_hidden.elf > cuckoo_hidden.c
```

This command will create the file **cuckoo_hidden.c** which will contain a representation of the memory map of the execution with C structures. This code gives us a more intuitive way of reading the data of the elf, e. g.:

```
/* data */
  "La flag es flag{*************}\n\0Gracias por portarte bien :)\n\0Muy mal"
    " hecho :'(\n",
```

Where we can find all the explicit strings stored in the elf file. Also, we can obtain the symbol table and its structura like:

```
/* flag */
    { 27, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      ADDR_TEXT + offsetof(elf, data), 0 },
    /* gracias */
    { 9, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      ADDR_TEXT + offsetof(elf, data) + 0x20, 0 },
    /* mal */
    { 17, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      ADDR_TEXT + offsetof(elf, data) + 0x3E, 0 },
    /* print_flag */
    { 21, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text), 0 },
    /* read_line */
    { 32, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0x18, 0 },
    /* read_loop */
    { 42, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0x31, 0 },
    /* not_bad */
    { 52, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xA3, 0 },
    /* sigue */
    { 60, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB0, 0 },
    /* len */
    { 66, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB2, 0 },
    /* len_start */
    { 70, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB9, 0 },
    /* print */
    { 80, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xCB, 0 },
```

Where **SHN_DATA** represents explicit strings and SHN_TEXT represents the names of the methods used in the program..

cuckoo_hidden.c

```
#include <stddef.h>
#include <elf.h>

#define ADDR_TEXT 0x00400000

enum sections
{
  SHN_TEXT = 1, SHN_DATA, SHN_SYMTAB, SHN_STRTAB, SHN_SHSTRTAB, SHN_COUNT
};

typedef struct elf
{
  Elf64_Ehdr      ehdr;
  Elf64_Phdr      phdrs[3];
  unsigned char   pad1[3864];
  unsigned char   text[256];
  unsigned char   pad2[3840];
  unsigned char   data[81];
  unsigned char   pad3[7];
  Elf64_Sym       symtab[18];
  char            strtab[110];
  char            shstrtab[39];
  unsigned char   pad4[3];
  Elf64_Shdr      shdrs[SHN_COUNT];
} elf;

elf foo =
{
  /* ehdr */
  {
    { 0x7F, 'E', 'L', 'F', ELFCLASS64, ELFDATA2LSB, EV_CURRENT, ELFOSABI_SYSV,
      0, 0, 0, 0, 0, 0, 0, 0 },
    ET_EXEC, EM_X86_64, EV_CURRENT, ADDR_TEXT + offsetof(elf, text) + 14,
    offsetof(elf, phdrs), offsetof(elf, shdrs), 0, sizeof(Elf64_Ehdr),
    sizeof(Elf64_Phdr), sizeof foo.phdrs / sizeof *foo.phdrs,
    sizeof(Elf64_Shdr), sizeof foo.shdrs / sizeof *foo.shdrs, SHN_SHSTRTAB
  },
  /* phdrs */
  {
    { PT_LOAD, PF_R, 0, ADDR_TEXT, ADDR_TEXT, offsetof(elf, pad1),
      offsetof(elf, pad1), 0x1000 },
    { PT_LOAD, PF_R | PF_X, offsetof(elf, text),
      ADDR_TEXT + offsetof(elf, text), ADDR_TEXT + offsetof(elf, text),
      sizeof foo.text, sizeof foo.text, 0x1000 },
    { PT_LOAD, PF_R | PF_W, offsetof(elf, data),
      ADDR_TEXT + offsetof(elf, data), ADDR_TEXT + offsetof(elf, data),
      sizeof foo.data, sizeof foo.data, 0x1000 }
  },
  /* pad1 */
```

```
{ 0 },
/* text */
{
  0x48, 0x8D, 0x34, 0x25, 0x00, 0x20, 0x40, 0x00, 0xE8, 0xBE, 0x00, 0x00,
  0x00, 0xC3, 0xE8, 0x05, 0x00, 0x00, 0x00, 0xE9, 0xDC, 0x00, 0x00, 0x00,
  0x55, 0x68, 0xFE, 0xCA, 0x00, 0x00, 0x48, 0x89, 0xE5, 0x48, 0x81, 0xEC,
  0x02, 0x01, 0x00, 0x00, 0x66, 0xC7, 0x85, 0xFE, 0xFE, 0xFF, 0xFF, 0x00,
  0x00, 0xB8, 0x00, 0x00, 0x00, 0x00, 0xBF, 0x00, 0x00, 0x00, 0x00, 0x48,
  0x31, 0xD2, 0x66, 0x8B, 0x95, 0xFE, 0xFE, 0xFF, 0xFF, 0x48, 0x8D, 0xB4,
  0x15, 0x00, 0xFF, 0xFF, 0xFF, 0x48, 0x31, 0xD2, 0xBA, 0x01, 0x00, 0x00,
  0x00, 0x0F, 0x05, 0x8A, 0x16, 0x66, 0x31, 0xC9, 0x66, 0x8B, 0x8D, 0xFE,
  0xFE, 0xFF, 0xFF, 0x66, 0xFF, 0xC1, 0x66, 0x89, 0x8D, 0xFE, 0xFE, 0xFF,
  0xFF, 0x80, 0xFA, 0x0A, 0x75, 0xBF, 0x66, 0x31, 0xD2, 0x66, 0x8B, 0x95,
  0xFE, 0xFE, 0xFF, 0xFF, 0xC6, 0x84, 0x15, 0xFD, 0xFE, 0xFF, 0xFF, 0x00,
  0x48, 0x81, 0xC4, 0x02, 0x01, 0x00, 0x00, 0x5A, 0x81, 0xFA, 0xFE, 0xCA,
  0x00, 0x00, 0x74, 0x0F, 0x48, 0x8D, 0x34, 0x25, 0x3E, 0x20, 0x40, 0x00,
  0xE8, 0x2A, 0x00, 0x00, 0x00, 0xEB, 0x0D, 0x48, 0x8D, 0x34, 0x25, 0x20,
  0x20, 0x40, 0x00, 0xE8, 0x1B, 0x00, 0x00, 0x00, 0x5D, 0xC3, 0x55, 0x48,
  0x89, 0xE5, 0x48, 0x31, 0xD2, 0x48, 0xFF, 0xC2, 0xB3, 0x00, 0x3A, 0x1C,
  0x16, 0x75, 0xF6, 0x48, 0x31, 0xC0, 0x48, 0x8D, 0x02, 0x5D, 0xC3, 0x55,
  0x48, 0x89, 0xE5, 0x48, 0x83, 0xEC, 0x10, 0xE8, 0xDA, 0xFF, 0xFF, 0xFF,
  0x48, 0x89, 0x45, 0xF0, 0xB8, 0x04, 0x00, 0x00, 0x00, 0xBB, 0x01, 0x00,
  0x00, 0x00, 0x48, 0x89, 0xF1, 0x8B, 0x55, 0xF0, 0xCD, 0x80, 0x48, 0x83,
  0xC4, 0x10, 0x5D, 0xC3, 0xB8, 0x01, 0x00, 0x00, 0x00, 0xBB, 0x00, 0x00,
  0x00, 0x00, 0xCD, 0x80
},
/* pad2 */
{ 0 },
/* data */
"La flag es flag{*************}\n\0Gracias por portarte bien :)\n\0Muy mal"
  " hecho :'(\n",
/* pad3 */
{ 0 },
/* symtab */
{
  { 0, 0, 0, SHN_UNDEF, 0, 0 },
  /* the.asm */
  { 1, ELF64_ST_INFO(STB_LOCAL, STT_FILE), STV_DEFAULT, SHN_ABS, 0, 0 },
  /* flag */
  { 27, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
    ADDR_TEXT + offsetof(elf, data), 0 },
  /* gracias */
  { 9, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
    ADDR_TEXT + offsetof(elf, data) + 0x20, 0 },
  /* mal */
  { 17, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
    ADDR_TEXT + offsetof(elf, data) + 0x3E, 0 },
  /* print_flag */
  { 21, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
    ADDR_TEXT + offsetof(elf, text), 0 },
  /* read_line */
  { 32, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
    ADDR_TEXT + offsetof(elf, text) + 0x18, 0 },
    /* read_loop */
    { 42, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0x31, 0 },
    /* not_bad */
    { 52, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xA3, 0 },
    /* sigue */
    { 60, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB0, 0 },
    /* len */
    { 66, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB2, 0 },
    /* len_start */
    { 70, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xB9, 0 },
    /* print */
    { 80, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xCB, 0 },
    /* end */
    { 106, ELF64_ST_INFO(STB_LOCAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 0xF4, 0 },
    /* _start */
    { 73, ELF64_ST_INFO(STB_GLOBAL, STT_NOTYPE), STV_DEFAULT, SHN_TEXT,
      ADDR_TEXT + offsetof(elf, text) + 14, 0 },
    /* __bss_start */
    { 86, ELF64_ST_INFO(STB_GLOBAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      ADDR_TEXT + offsetof(elf, pad3), 0 },
    /* _edata */
    { 98, ELF64_ST_INFO(STB_GLOBAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      ADDR_TEXT + offsetof(elf, pad3), 0 },
    /* _end */
    { 105, ELF64_ST_INFO(STB_GLOBAL, STT_NOTYPE), STV_DEFAULT, SHN_DATA,
      0x402058, 0 }
  },
  /* strtab */
  "\0the.asm\0gracias\0mal\0print_flag\0read_line\0read_loop\0not_bad\0sigue"
    "\0len\0len_start\0print\0__bss_start\0_edata\0_end",
  /* shstrtab */
  "\0.symtab\0.strtab\0.shstrtab\0.text\0.data",
  /* pad4 */
  { 0 },
  /* shdrs */
  {
    { 0, SHT_NULL, 0, 0, 0, 0, SHN_UNDEF, 0, 0, 0 },
    /* .text */
    { 27, SHT_PROGBITS, SHF_EXECINSTR | SHF_ALLOC,
      ADDR_TEXT + offsetof(elf, text), offsetof(elf, text), sizeof foo.text,
      SHN_UNDEF, 0, 0x10, 0 },
    /* .data */
    { 33, SHT_PROGBITS, SHF_WRITE | SHF_ALLOC,
      ADDR_TEXT + offsetof(elf, data), offsetof(elf, data), sizeof foo.data,
      SHN_UNDEF, 0, 4, 0 },
    /* .symtab */
    { 1, SHT_SYMTAB, 0, 0, offsetof(elf, symtab), sizeof foo.symtab,
      SHN_STRTAB, 14, sizeof(Elf64_Addr), sizeof(Elf64_Sym) },
    /* .strtab */
    { 9, SHT_STRTAB, 0, 0, offsetof(elf, strtab), sizeof foo.strtab,
      SHN_UNDEF, 0, 1, 0 },
    /* .shstrtab */
    { 17, SHT_STRTAB, 0, 0, offsetof(elf, shstrtab), sizeof foo.shstrtab,
      SHN_UNDEF, 0, 1, 0 }
  }
};
```

overload_script.py

```
import argparse
import os

from numpy import char

# Command line arguments parsing
parser = argparse.ArgumentParser()
parser.add_argument("-f", "--file", help="Target file", required=True)
parser.add_argument("-n", "--number", help="Number of characters to overload",
required=True)
args = parser.parse_args()

# Create the overflow string
char_seq = "A" * int(args.number)

# If the target file exists, execute it and write 'char_seq' as its input
if (os.path.exists(args.file)):
    # Execute the file with char_seq as input
    os.system("echo " + char_seq + " | ./" + args.file)
else:
    print(char_seq)
    print("[!] File not found")
```