Software Vulnerabilities:
Exploitation and Mitigation
–
Lab 4

Prof. Jacques Klein & Pedro Jesús Ruiz Jiménez
(inspired from Prof. Alexandre Bartel's course)

4 Buffer Overflow on Non-Executable Stack (68 P.)

The goal is to exploit a buffer overflow on a non-executable stack to execute arbitrary code using ROP gadgets.

4.1 Setup Labs Environment

4.1.1 Update resources

From our *Git* directory (`SVEM`), use this command to update the lab resources:

```
$ git pull
```

4.2 Launch Emulated Environment

On your host machine, go to the `Lab04` directory found inside the cloned repository. Then, create the virtual machine (vm) and connect to it by using the following commands:

```
$ vagrant up
$ vagrant ssh
```

Other important vagrant commands that you might need are:

```
$ vagrant halt  # Stops vm
$ vagrant reload  # Resets vm
$ vagrant destroy  # Destroys vm
$ vagrant provision  # Rerun provision
$ vagrant global-status  # Status about vms
```

Finally, once you are connected to the vm, you'll need to move to the work directory: `lab`

4.3 Gadgets

Question 4.1 What is *python-capstone*? What is *python-elftools*? 4 P.

```
$ lscpu
```

Question 4.2 Use the lscpu command to check if *VirtualBox* emulating a 32 or 64 bit processor. What are the main differences between 32 and 64 bit processors? 4 P.

Question 4.3 Why is it important to know the processor architecture when disassembling native code? 2 P.

The following program is `gadgets.py`:

```python
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


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

```

```python

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

        execSections = []
        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"]
        checkedSections = [".init", ".plt", ".text", ".fini"]

        for nsec, section in enumerate(elffile.iter_sections()):

            # check if it is an executable section containing instructions

            # good sections we know so far:
            #.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

            if section.name not in goodSections:
                continue

            # add new executable section with the following information
            # - name
            # - address where the section is loaded in memory
            # - hexa string of the instructions
            name = section.name
            addr = section['sh_addr']
            byteStream = section.data()
            hexStream = binascii.hexlify(byteStream)
            newExecSection = {}
            newExecSection['name'] = name
            newExecSection['addr'] = addr
            newExecSection['hexStream'] = hexStream
            execSections.append(newExecSection)

        return execSections


if __name__ == '__main__':
    if sys.argv[1] == '--test':

```

```
        md = Cs(CS_ARCH_X86, CS_MODE_64)
        for filename in sys.argv[2:]:
            r = getHexStreamsFromElfExecutableSections(filename)
            print "Found ", len(r), " executable sections:"
            i = 0
            for s in r:
                print "   ", i, ": ", s['name'], "0x", hex(s['addr']), s['hexStream']
                i += 1

                hexdata = s['hexStream']
                gadget = hexdata[0 : 10]
                gadget = convertXCS(gadget)
                offset = 0
                for (address, size, mnemonic, op_str) in md.disasm_lite(gadget, offset):
                    print ("gadget: %s %s \n") %(mnemonic, op_str)
```

The program `gadgets.py` 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.

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 4.4 How many different ret instructions are there? (look on page 1784 of the official documentation) What are the differences? 4 P.

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

Question 4.5 What is the maximum size of an instruction? Find the answer by reading the complete official documentation [a]. At what page did you find the answer? 2 P.

[a] just kidding :-), search the pdf using the keywords "instruction-size limit" or "instruction length limit"

Question 4.6 Modify "gadgets.py" 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. Remember that gadgets do not contain branching instructions which may kill the gadget (you can filter them or print them, but printing them adds noise). Do not forget that all gadgets end with a "ret" instruction. 15 P.

Question 4.7 How many gadgets of length 1, 2 and 3 (length = number of instructions excluding the final "ret" instruction) are there in the `.text` section of the "/bin/ls" binary? 4 P.

You can compare the results of your gadget extractor with the ones of an existing tool. In the host machine, clone ROPGadget inside the `Lab02/lab` directory. Then, in the vm extract the list of gadgets using the following command:

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

Question 4.8 ROPgadget should at least find all the gadgets you found. How many gadgets of length 1, 2 and 3 that **you** generated are found by ROPgadget? How many gadgets of length 1, 2 and 3 that **you** generated are not found by ROPgadget? What is the purpose of the "–depth" option? Could this option impact the results of ROPgadget? 4 P.

4.4 Stack Overflow

The following C program is `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
  mbuffer = malloc(size);
  fread(mbuffer, size, 1, f);
  fclose(f);

  // copy content from the buffer on the heap
  // to the buffer on the stack
  memcpy(buffer, mbuffer, size);

  // dump hexa representation
  for (i = 0; i < MAX_BUFFER_SIZE; i++) {
    printf("%02X ", buffer[i]);
    if ((i+1) % 8 == 0) printf("    ");
    if ((i+1) % 16 == 0) printf("\n");
  }
  printf("\n");

  return 0;
}

int main(int argc, char** argv) {

  if (argc != 2) {
    printf("error: the first argument must \
        be the path to the file to hexdump.\n");
    return -1;
  }

  function2(argv[1]);

  // everything went according to plan
  return 0;
}
```

Compile `hexdump.c` using the following command to make the stack non-executable:

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

The input file `input1.dat` contains the text: `Hi there!`. Run the program using the following command:

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

Question 4.9 What is the output? What does the program `hexdump` do? 3 P.

Question 4.10 Where and what is the security vulnerability in this program? 2 P.

Question 4.11 What input should you give to the program to generate a *Segmentation Fault*? 2 P.

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

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

Question 4.12 List the libraries having an executable section linked to the program. 3 P.

Question 4.13 Use your "gadgets.py" program to compute gadgets of length 1, 2 and 3 for the `libc` library and the `hexdump` program. How many gadgets of length 1, 2 and 3 do you get for `libc`? How many gadgets of length 1, 2 and 3 do you get for `hexdump`? 3 P.

At this point you have to find the appropriate gadgets from this list to simulate the execution of a shellcode. Recall that, in Lab 3, you used `syscall 59` (the execve function) to execute a shell. 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

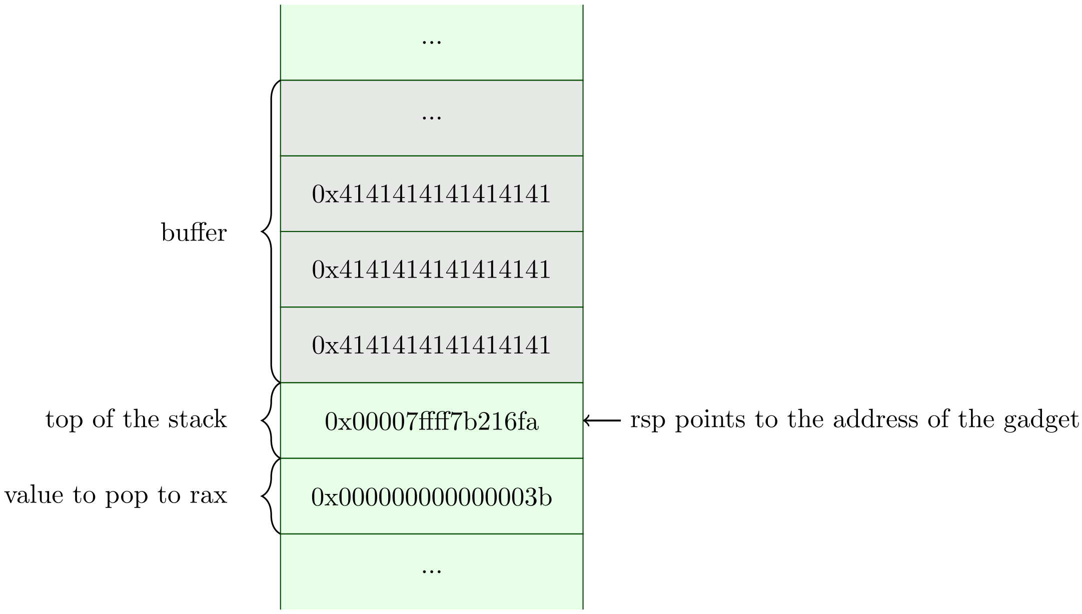
- `rdx` initialized with NULL

Do not forget to initialize `rax` with 59, since this is the register that the `syscall` instruction checks to know which method to execute.

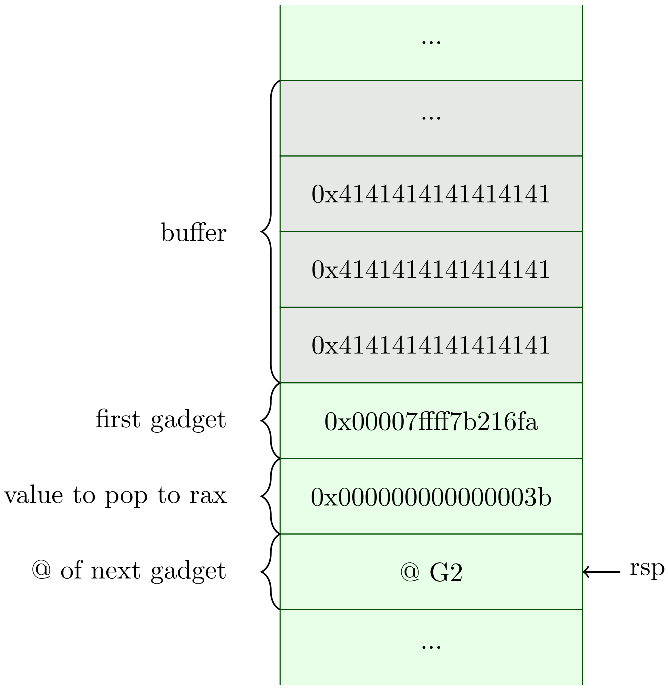
Let's start with the initialization of `rax`. The following gadget in libc allows us to pop a 64-bit value from the stack and store it in `rax`:

```
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 4.14 Which gadget did you use to initialize `rsi`? Explain. 2 P.

Question 4.15 Which gadget did you use to initialize `rdx`? Explain. 2 P.

Question 4.16 Which gadget did you use execute the `syscall` instruction? Explain. 2 P.

Question 4.17 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 4.18 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 cite the source from which you obtained the text.