General terminal commands:

Disable ASLR on your system until next reboot: **echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**
Enable ASLR on your system again: **echo 2 | sudo tee /proc/sys/kernel/randomize_va_space**
gives information about the file, e.g. 32 bit vs. 64 bit **file <binary>**

Basic Steps

STEP 1	How many Bytes to overwrite the Buffer until RIP? gdb -> pattern create
STEP 2	Base address of libc: gdb <binary> gdb-peda$ run ctrl+c (stop execution) gdb-peda$ vmmap Offset system: readelf -s /path/to/libc | grep system Offset /bin/sh: strings -tx /path/to/libc | grep /bin/sh
STEP 3	Find ROP gadget “pop rdi”: ROPgadget --binary <binary> | grep ”pop rdi”
STEP 4	Calculate absolute address of system Calculate absolute address of /bin/sh
STEP 5	Assemble payload: - Fill up the buffer (write number of bytes of STEP 1) - addresses of gadgets you want to jump to - addresses with p64()
STEP 6	Test your exploit locally: ./create-payload.py > payload.bin cat payload.bin - | ./02_demo if it does not work, debug it! Set the breakpoint on return! gdb <binary> gdb-peda$ break *main+xx (set breakpoint on return (disas main)) gdb-peda$ run < payload.bin
STEP 7	Test your exploit remote: cat payload.bin - | ncat <ip-addr> <port>

pwntools:

from pwn import *	to use pwntools in python	
p64(<integer>)	convert 64 bit integer to little endian bytestring	p64(0x7fabc)

ROPgadget:

```
ROPgadget --binary <binary>
```

Command line tricks:

store a payload that spawns a shell into a file, and provide it as input to the vulnerable binary and keep stdin open so the shell does not exit:

```
./exploitscript.py > payload.bin
cat payload.bin - | ./01_exercise
```

gdb / peda

`disas <function>`	Disassembles code	`disas main`

`break`
`b`
	Sets a breakpoint
	- when debugging your exploit, set the breakpoint on return!
		`break *main+117`
		`b     *main+117`

`run`
`run < <input-file>`
	runs the binary	`run`
		`run < payload.bin`

`ctrl+c`	Stops the execution	
`c`	continue execution until next stop	
`ni`	“next instruction”, next instruction line (steps over function calls)	
`si`	“step into”, next instruction, but steps into function calls	
`checksec`	Shows which security features are turned on/turned off	

`vmmap`	Shows memory mapping (during execution)	`run`
		`break with ctrl+c`
		`vmmap`

`aslr on`	Turns aslr in gdb on	
`pattern create <number>`		`pattern create 70`
`pattern offset <pattern>`	Take the pattern you find in RSP (64 bit: RIP does not load the overflown pattern, take RSP)	`pattern offset AA(A`

Important addresses and offsets inside a binary or the libc:

Libc Base	gdb-peda	⇒ run ⇒ ctrl + c ⇒ **vmmap**
Offset system	Command line	**readelf -s** /path/to/libc **| grep system**
Offset "/bin/sh"	Command line	**strings –tx** /path/to/libc | **grep /bin/sh**

Ghidra:

`File → New Project → Non-shared Project → Project Name <Your Project> → Finish`	New Project
`File → Import File → <Your File>`	Add binary to project
`DoubleClick on imported File`	Open imported binary

`Find Functions (like e.g. main function):`
`Symbol Tree (left sidebar) → Functions → main`	