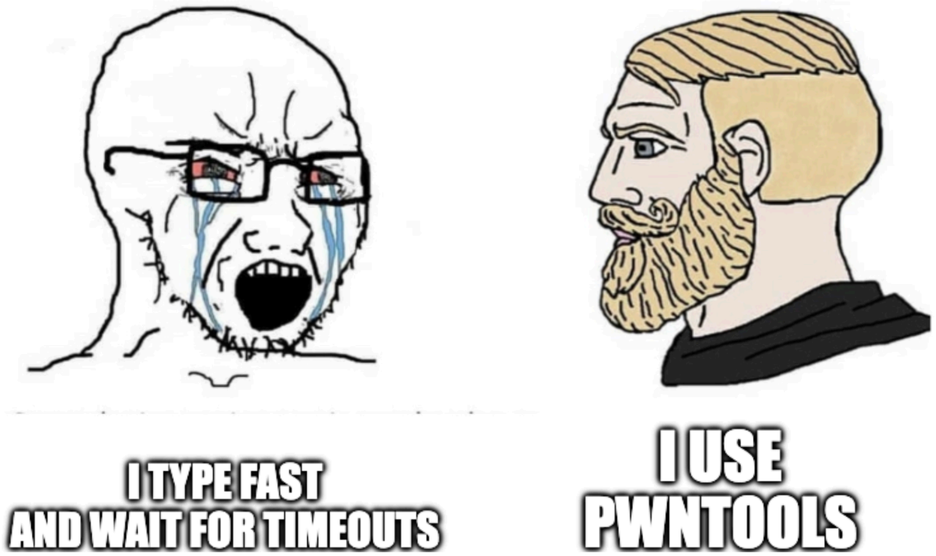
Writing Exploits

Using Python and pwntools

Motivation

But why?

- More advanced exploits might need runtime specific information
  - Addresses, offsets, magic values etc.
- We want on interface for:
  - Run time interaction with executable / remote
  - Help with analysis, gadget finding (see later talk) etc.



PWNTOOLS

PWNTOOLS

- CTF exploit development framework
- See git for all the things: https://github.com/Gallopsled/pwntools
- https://github.com/Gallopsled/pwntools-tutorial/
- Set up virtual environment / **uv (ask Merlin)**

```
python3 -m venv ~/ctf-venv
source ~/ctf-venv/bin/activate
```

- Install pwntools

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade pwntools
```

Tubes

Local, Remote, SSH

- Tubes are pwntool's way of managing communication
- Processes:

```
io = process(['sh', '-c', 'echo $MYENV'], env={'MYENV': 'MYVAL'})
io.recvline()
```

- Remote

```
io = remote('google.com', 80)
io.send('GET /\r\n\r\n')
io.recvline()
```

```
session = ssh('bandit0', 'bandit.labs.overthewire.org', 2220, password='bandit0')
io = session.process('/bin/sh', env={“PS1”:""})
io.sendline('echo Hello, world!')
io.recvline()
```

```
client = listen(8080).wait_for_connection() # Binds to localhost
```

Context

Defining the run environment

- Context allows to set the “environment” for the binary
- Adjusts the output of other commands according to what is required & specified

```
from pwn import *

context.arch = 'amd64'
context.endian = 'little'
context.log_level = 'debug'
context.timeout = 5

context.update(arch='amd64', endian='little', log_level='debug', timeout=5
```

Packing, Encoding and Decoding

Defining the run environment

- Influenced by context!!

```
from pwn import *
context.endian = 'little'
little = p64(0x41)
context.endian = 'big'
big = p64(0x41)

# Little: b’A\x00\x00\x00\x00\x00\x00\x00',
# Big: b'\x00\x00\x00\x00\x00\x00\x00A'
```

- Encoding and decoding things:

```
enhex('hello')
# '68656c6c6f'
unhex('776f726c64')
# ‘world'
hexdump(read('/dev/urandom', 32
```

```
urlencode("Hello, World!") == ‘%48%65%6c%6c%6f%2c%20%..‘

‘hello' == b64d(b64e('hello'))

unpack('AAAA') == 0x41414141 # unpacks to integer
pack(0x41414141) == b'AAAA' # packs to bytes
```

ELF Utilities

- We can also use pwntools to analyze ELF files

```
e = ELF('/bin/bash')
# We can retrieve addresses from non-stripped binaries
e.symbols  ['bash_license']
e.symbols  ['execve']
e.got      ['execve']
e.plt      ['execve']
e.functions['list_all_jobs']
```

- **address** and **entry** contain the base address and the entrypoint

- Even provides support for packing your own ELF binaries if you so desire

ROP'ing and other stuff

Helperlings

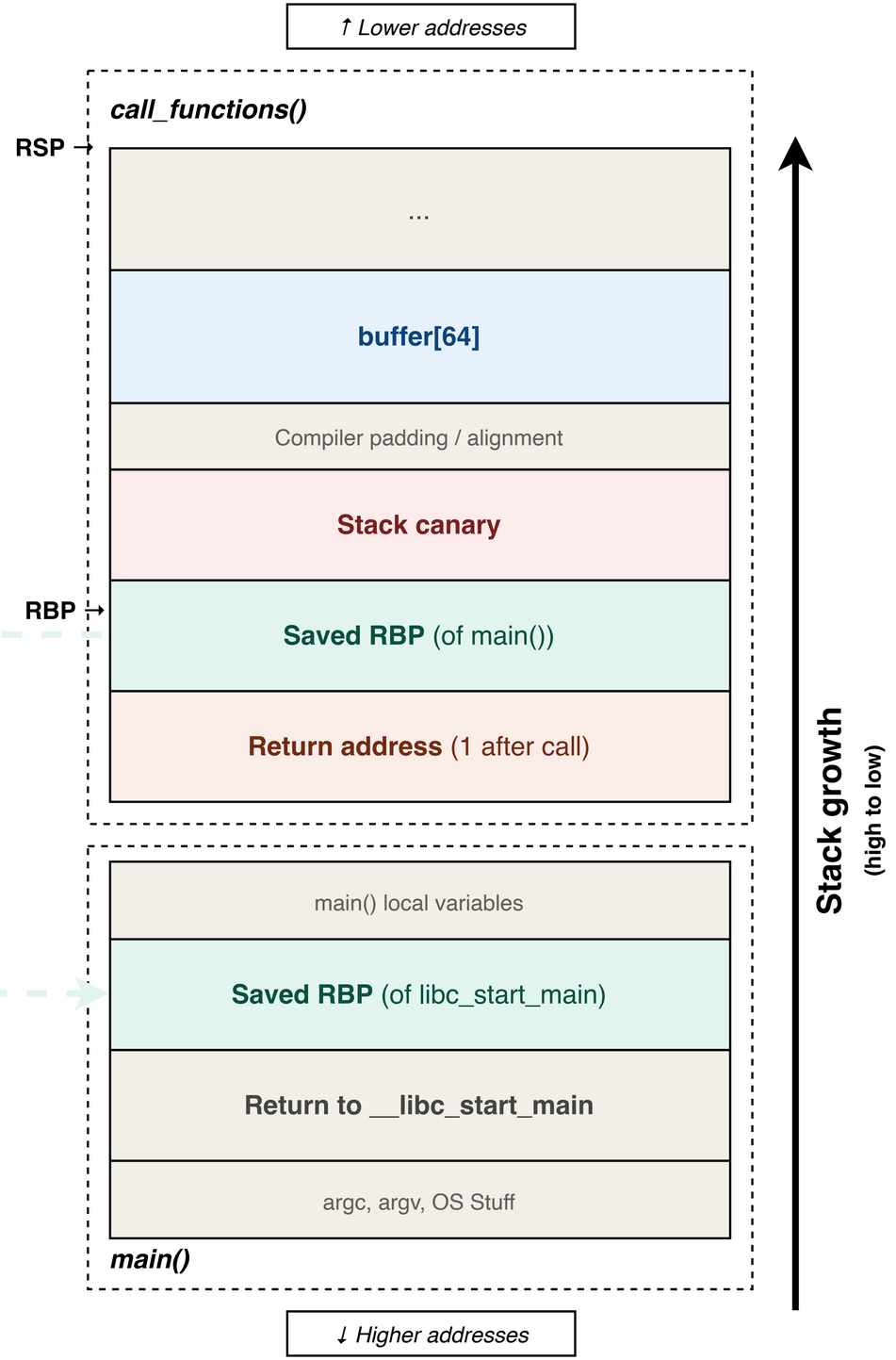
- Eventually we will be trying to do R(eturn) O(riented) P(rogramming)
- Keep these in mind:

```
elf = ELF('/bin/sh')
rop = ROP(elf)

rop.rbx # Cleanest version
# Gadget(0x5fd5, ['pop rbx', 'ret'], ['rbx'], 0x8)

rop.gadgets
# {4278278088: Gadget(0xff0157c8, ['add esp, 0x130', 'pop rbp', 'ret'], ['rbp'], 0x138),
#  4278214225: Gadget(0xff005e51, ['pop rsp', 'ret'], ['rsp'], 0x8),
#  4278210586: Gadget(0xff00501a, ['ret'], [], 0x4)}
# ...
```

PIETIME2


↑ Lower addresses
call_functions()
RSP
...
buffer[64]
Compiler padding / alignment
Stack canary
RBP
Saved RBP (of main())
Return address (1 after call)
main() local variables
Saved RBP (of libc_start_main)
Return to __libc_start_main
argc, argv, OS Stuff
main()
↓ Higher addresses
Stack growth
(high to low)