Explaination of ➔ how the `rip` argument is passed and used in the code, even though it seems like `rip` is not explicitly set within the function.

Understanding `rip` in Context

In the script, `rip` is an argument in the `send_payload()` function. Here’s how it's handled:

```
def send_payload(rip):
    payload = b"A" * offset + struct.pack("<Q", rip) + b"\n"
```

- **`rip`** stands for "return instruction pointer," and it refers to the address where we want the program to jump to after the buffer overflow occurs.
- **`rip` is passed as an argument** when you call the function `send_payload(flag_address)` at this line:

```
response = send_payload(flag_address)
```

In this case, **`flag_address` is the value 0x40123B** (the address of the `flag()` function), which is passed as the `rip` argument. So, inside the function `send_payload()`, `rip` is not undefined; it is **set to `flag_address`**, which contains the address we want to overwrite the return pointer with.

Packing the RIP Value

Inside the function:

```
payload = b"A" * offset + struct.pack("<Q", rip) + b"\n"
```

- **`struct.pack("<Q", rip)`** converts the `rip` value (in this case, the `flag_address`) into a **64-bit little-endian format**, which is required to match the memory architecture of the target system (64-bit).
- This packed `rip` is then appended to the payload, which overflows the buffer and alters the return address to point to the `flag()` function.

Key Point:

Even though `rip` is not directly defined inside the function, it is set when the function is called, and the value passed (`flag_address`) is used within the function. This is standard in Python function arguments — you pass a value when calling the function, which the function then uses internally.