Question 5.1 - What is BROP? What is the difference between BROP and ROP? (2-3 sentences)
BROP (Blind Return Oriented Programming) is a new technique, proposed by the paper, which allows for exploiting software vulnerabilities when both the target binary and source code are unknown. It focuses on finding Return Oriented Programming (ROP) gadgets remotely to construct exploits. BROP works blindly without prior knowledge of the target binary or the source code, while traditional ROP requires knowledge of the binary's structure to build the exploit.

Question 5.2 - Does a BROP attack also work for binaries for which the attacker does not have access to (no source, no binary)? Explain. (3-4 sentences)
Yes, BROP is specifically designed to exploit proprietary services without prior knowledge or access of the target binary's structure. BROP remotely finds enough gadgets to perform the *write* system call, after which the server's binary can be transferred from memory to the attacker's socket. At this point, canaries, ASLR and NX have been defeated and the exploit can proceed using known techniques.

Question 5.3 - Look at Figure 4. Is there a situation where the BROP approach does not work? Why? (3-4 sentences)
Yes, the BROP attack can't target PIE servers that rerandomize (e.g., *execve*) after a crash. PIE (Position Independent Executable) is a security feature that randomizes the location of the loaded binary code at runtime. Rerandomization is a specific PIE implementation where the address space is randomized (i.e., the memory locations are changed) every time the program crashes. As a result, BROP cannot locate and exploit ROP gadgets remotely on the target system through trial and error.

Question 5.4 - What are the requirements to perform a BROP attack? Explain. (3-4 sentences)
The attacker needs a stack vulnerability that can be triggered, and knowledge of how to exploit it. The attacker further requires a server application that restarts after a crash in order to maintain the attack cycle. The threat model assumes that the attacker can crash the server multiple times in order to overwrite a variable length of bytes, including a return instruction pointer. The threat model further assumes that the attacker doesn't know nor need to know the source code or target binary of the server.

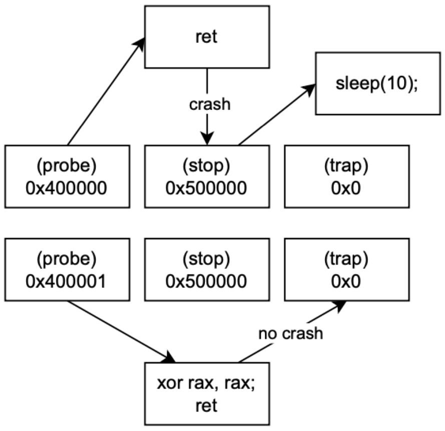
Question 5.5 - Instead of randomly brute-forcing the 2^{64} address space, an attacker can leak the return address. How does an attacker leak the return address? Explain. (3-4 sentences)
An attacker can leak the return address by using a technique known as stack reading. This method involves overwriting the stack byte-by-byte with potential guess values (also known as canaries) until the correct one is found, causing the server not to crash. By repeatedly overflowing the stack (128 tries of the algorithm on average) and observing the server's response, the attacker can determine the correct return address, allowing him to bypass Address Space Layout Randomization (ASLR) on 64-bit systems.

Question 5.6 - What are stop gadgets? What are they useful for? Explain. (3-5 sentences)
Stop gadgets are instructions or code snippets that cause the program to block, such as infinite loops or blocking system call (like sleep). They are essential for halting the execution of a ROP chain to prevent crashes and enable the detection of gadgets. Stop gadgets serve as signalling mechanisms, allowing attackers to determine whether a specific gadget executed or not. This helps with the identification and exploitation of useful gadgets within the program.

Question 5.7 - To classify gadgets, different stack layouts (the data the attacker puts on the stack) are used. What are the different stack layouts and how can they be used to identify categories of gadgets? Explain. (5-10 sentences + figure)
By varying the position of the stop and trap on the stack, the attacker can deduce the instructions being executed by the gadget. The presence or absence of crashes helps classify gadgets based on their behavior (such as stack interaction patterns).

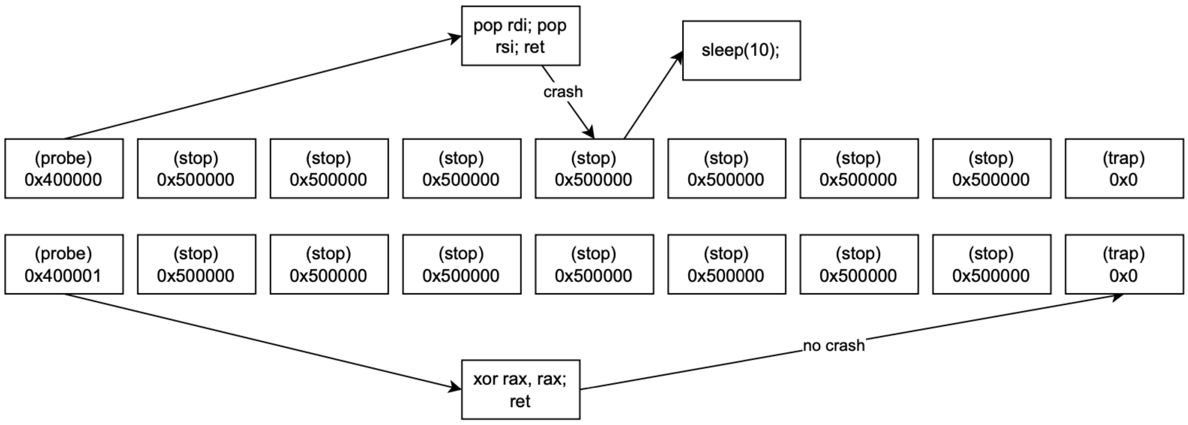
The examples of possible stack layouts in the paper are the following:

- **probe, stop, traps (trap, trap, . . .):** Finds gadgets that do not pop the stack like *ret* or *xor rax, rax; ret.*



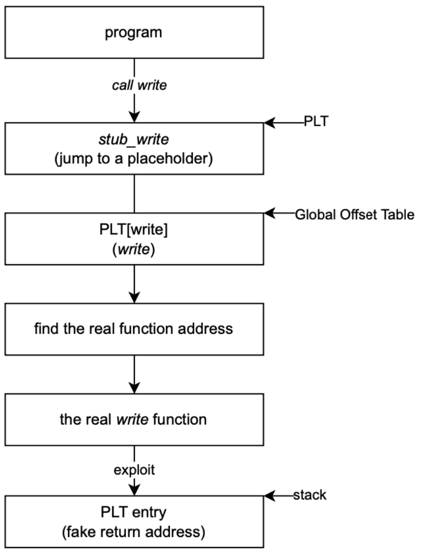
- **probe, trap, stop, traps:** Finds gadgets that pop exactly one stack word like *pop rax; ret* or *pop rdi; ret*. Figure 10 in the paper shows an illustration of this.

- **probe, stop, stop, stop, stop, stop, stop, stop, traps:** Finds gadgets that pop up to six words (e.g., the BROP gadget).



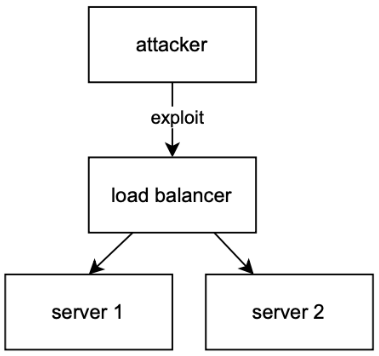
Question 5.8 - To execute a function, the attacker has to find the PLT. What is the PLT? How can an attacker find this table? Explain. (5-10 sentences + figure)

The PLT (Procedure Linking Table) is a jump table at the beginning of the executable used for all external calls (e.g., libc). It is used for dynamic linking containing all external library calls made by the application. The attacker can locate the PLT by scanning from the program's origin (0x400000) or backwards from an address leaked through stack reading if the PIE flag was used. Each address must be 16 bytes aligned, and 16 bytes can be skipped per probe for efficiency. The attacker can verify the PLT by ensuring neighboring entries do not crash and that offsets of six bytes (the PLT slow path) do not cause a crash. The stack layout to find a PLT entry will be: probe, stop, trap.

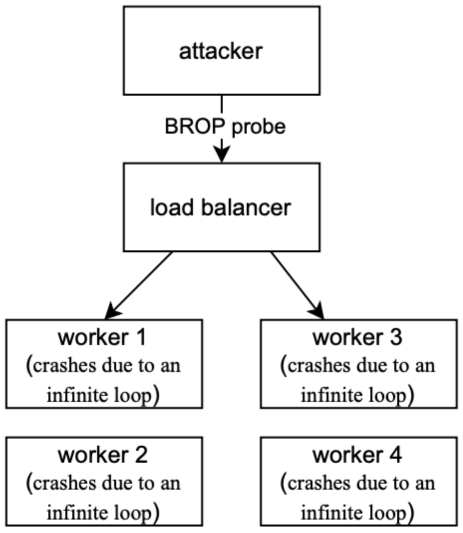

program
call write
stub_write (jump to a placeholder)
PLT
PLT[write] (write)
Global Offset Table
find the real function address
the real write function
exploit
PLT entry (fake return address)
stack

Question 5.9 - How long, did it take to exploit a vulnerability with a BROP attack in yaSSL + MySQL? in nginx?
It took approximately 4,000 requests or 20 minutes to exploit a vulnerability with a BROP attack in yaSSL + MySQL. It took 1 minute to exploit a vulnerability with a BROP attack method in nginx.

Question 5.10 - The return address might not be recovered if a load balancer is used (we assume here attack on a remote host through the network). Explain why. (5-10 sentences + figure)
One of the limitations of this paper and the BROP attack is the load balancer, as it can cause the attack to fail when PIE is used and canaries can't be circumvented. When attacking a remote host through the network, the return address recovery process relies on manipulating the stack to control program flow. Load balancers distribute incoming network traffic across multiple servers to optimize resource utilization and prevent server overload. Thus, the network path may change, altering the memory layout and stack contents. This makes it difficult to consistently target and recover the return address. The attacker's controlled byte could also be overwritten or modified during distribution transit. Furthermore, analyzing a crash on the target system becomes difficult because the attacker doesn't know which server crashed, as he only sees the load balancer.



Question 5.11 - The attack might not work if all workers are stuck in a loop. Explain why. (5-10 sentences + figure)
Another limitation of this paper and the BROP attack is that the attack relies on a number of workers being available and not being stuck in an infinite loop. If all workers become stuck in an infinite loop, the attack flow is disrupted as the workers are unable to handle new connections or process incoming requests. Stuck workers still consume resources even though they are not completing tasks. This leads to resource exhaustion and potential denial of service for legitimate requests. Furthermore, the crashes caused by the BROP probes wouldn't be distinguishable from the crashes caused by the infinite loop. This makes identifying the ROP gadgets more difficult. Thus, the exploitation process could be halted, which prevents the attacker from manipulating the stack and executing attacks successfully.


attacker
BROP probe
load balancer
worker 1 (crashes due to an infinite loop)
worker 3 (crashes due to an infinite loop)
worker 2 (crashes due to an infinite loop)
worker 4 (crashes due to an infinite loop)