1/10) Is the program leak.c vulnerable to a buffer overflow of type stack smashing/control hijacking? (If yes why, if no why)

No, the program is not vulnerable to a buffer overflow attack involving the stack, because the buffer is allocated on the heap, through the use of the malloc function.

2/10) Is the program leak.c vulnerable to an attack of type heap manipulation? (If yes why, if no why)

Yes, it is. The program allocates memory on the heap using the malloc function, but does not free it before returning. As a result, this memory remains allocated and inaccessible after the isThisVulnerable() function returns. If the function is called repeatedly without freeing the previously allocated memory, it can lead to memory leaks and eventually cause the program to run out of memory. Additionally, the program does not perform any bounds checking on the size parameter before allocating memory using malloc. This can allow an attacker to pass a large value for size that causes the program to allocate more memory than intended. If the program is not designed to handle such large allocations, it can lead to a DoS attack by causing the program to run out of memory or crash. To mitigate these vulnerabilities, the program should free any allocated memory before returning, and perform bounds checking on the size parameter to prevent malicious or accidental allocation of large amounts of memory.

3/10) Is the program leak.c vulnerable to an attack of type stack information exfiltration? (If yes why, if no why)

No, leak.c is not vulnerable to an attack of type stack information exfiltration, because it does not use or manipulate the program's stack. The program allocates memory on the heap using malloc and initializes it with the memset function. The buf array declared in the main function is also allocated on the stack, but it is only used to initialize the memory allocated, and is not accessed or modified afterwards. Therefore, there is no sensitive information stored on the stack that could be exfiltrated using the program.

4/10) Is the program leak.c vulnerable to an attack of type heap information exfiltration? (If yes why, if no why)

Yes, the program allocates memory on the heap using malloc and initializes it with the memset function. However, it does not overwrite the memory before returning, which means that any sensitive information that was previously stored in that memory may still be present in the heap. If an attacker can gain access to this memory, they may be able to extract this sensitive information. One way an attacker could gain access to the heap memory is by exploiting a buffer overflow vulnerability. If the program writes beyond the bounds of a heap-allocated buffer, it may overwrite the memory that follows it in the heap. If the overwritten memory contains sensitive information, the attacker may be able to read this information by allocating a buffer that overlaps with the overwritten memory.

5/10) Is the program leak.c vulnerable to any of the previously mentioned attacks if stack protectors (canaries) are in place? (If yes why, if no why)

No, as we said, since there is no buffer allocated on the stack, there is no vulnerability to a stack-based buffer overflow to start with. There is no need for stack protection mechanisms such as canaries or stack guards in this program.

6/10) Can not executable stack prevent the return loop in loop3.c? (If yes why, if no why)

No, the return loop is the result of the program overwriting the return address on the stack frame of the function that is executing. It's not executing code from the stack, like a shellcode, for example. Therefore, there is no code to be prevented from being executed in the stack.

7/10) Can stack protectors (canaries) prevent the return loop in loop3.c? (If yes why, if no why)

No. A canary is a security mechanism that involves placing a random value on the stack before the return address. This value is checked before a function returns, and if it has been modified, then an attacker has likely overwritten the return address and the program terminates. For the code, though, the memory location at the offset value 4 that we are overwriting, is not related to the canary (that would be located on the offset value 1), so canaries couldn't do anything to prevent the loop from happening.

8/10) Can ASLR prevent the return loop in loop3.c? (If yes why, if no why)

No, because ASLR shuffles randomly the addresses of the stack and heap but in this case, the address of the loop is taken at runtime, so it's not effective.

Let's make the Hypothesis that there is no "system" function call in the libc library
9/10) Is it possible to make a return to libc attack that activates any program that is present on the system? (If yes why, if no why, are there any restrictions)

Yes, as in the libc we can still find the exec() family of functions and they can be used to execute other programs on the system, including a shell. Therefore, it is possible to use a return-to-libc attack to execute arbitrary programs or shell commands by redirecting the program's control flow to the appropriate exec() function, e.g. execve().

10/10) Does ASLR protect from a return to libc attack? (If yes why, if no why)

Address Space Layout Randomization (ASLR) is a security feature that randomizes the memory addresses of program components at runtime. The purpose of ASLR is to make it more difficult for attackers to predict the memory address of a particular function or gadget and use it to launch an exploit. ASLR can make it more difficult to carry out a return-to-libc attack, but it does not completely protect against such attacks. While ASLR can randomize the memory addresses of the libc library and other program components, it does not prevent an attacker from finding gadgets within those components that can be used to construct an exploit. Moreover, with a 32-bit system, the attacker can still be able to brute-force his way in around 19 minutes. Nevertheless, with ASLR enabled, the attacker needs to discover the randomized addresses of the necessary gadgets and functions within the libc library, which can be really difficult.