CS155 PS1
Yang Du

Problem 1:

a. There are two copies of the return address, the attacker have to overwrite both copies in order to prevent the program exit. When the attacker wants to overflow the string to change the return address, he cannot exactly know in advance where to override the random copy of the return address. Thus, it is obviously harder for the buffer overflow attack.
b. int bar(char *input)
   {
   void (*foo)() = &myFunc;
   char buf [1024];
   strcpy(buf, input);
   foo();
   return 0;
   }

   myFunc can be any function.
   When the attacker call this function bar, with input having attacking code pointer. The function pointer will be overwritten during strcpy. When calling foo, the attacker code will be executed.

Problem 2:

0x40044444	eip
0x40002948	eip
0x41000000	stack
0x40002700	eip
0x40000000	stack
0x40002000	ret-addr
anything	SFP
String buffer	

In this stack, it will first run the code pop eax, which pop 0x40000000 to the register eax, then, it will return to the code pop ebx, which pop 0x41000000 to the register ebx, then, it will return to the code move [ebx], eax, which write the value of eax to location ebx. At last, it will return to the address 0x40044444, which is the return address the attacker wants to go.

Problem 3:
The line is used to prevent integer overflow. If there is no check for this, if size if a very large number, and sizeof(*hdr) is 4 bytes as a pointer usually be, then the memsize might overflow to another integer such as a negative number, zero, or something positive but very small. Then the attacker can utilize this to exploit. For example, an attacker can set the size to a very large number, and when add 4 bytes

of hdr pointer, the memsize will be overflow to something like 2 bytes. Then, only two bytes of memory will be allocated. When hdr -> mlen = memsize, a 4 bytes int size_t will be write to the memory address pointed by hdr. Thus, two bytes in memory are already overwritten. If the attacker wants to overwrite the two byte, this is vulnerability. However, the number being overwritten might be limited to very small number(less than sizeof(*hdr)). The attacker might call the malloc function again, and make the size to be very large. Since the system only give two bytes to the previous malloc, this malloc will start 2 bytes after the previous hdr points, thus, it can overwrite the third and the fourth bytes of the previous length to be a large number. Later, when the attacker call the free() function, the size has already been change to some large number, so that a large amount of memory will be free instead of the originally 2 bytes. The attacker can do many bad things with this vulnerability.

In addition, if the system only have two bytes of free memory, and the size is overflow to 2, then the malloc func will successfully return, however, actually the system doesn't have enough memory for the mhead, and it is another vulnerability.

Problem 4:

a. The tool can be sound, but it cannot be complete, because a complete tool will not report any errors if the program does not contain an error, so that false alarm is impossible to occur when running a complete tool. But we are not sure if the tool detects all the error, so that the tool might be sound.
b. Soundness. The company needs to have a tool, which is soundness because it wants to detect all the security vulnerabilities before releasing their product. The soundness property of the tool guarantees to report all the error. Thus, soundness property is essential.
c. Because the tool is both sound and complete, it will report all the errors, and it will not have false alarm, thus, if the loop doesn't terminate, which is the exit condition for the loop never meet, for example, there is a return statement in the loop, or the loop is an infinite loop, the tool will not report an error. Otherwise the tool will report an error when the loop exits.

Problem 5:

a. The if statement checks d.idx > dma -> buf_count, thus, if d.idx hits an upper bound, the program will return. Thus, after the first line, it will have an upper bound checking. However, since it only check the upper bound, not the lower bound, it will remain tainted.
b. Since d.idx is still tainted, if it involved the array[v], etc. activities, it will report an error. Thus, an error will be generated because a tainted variable is doing array.
c. The variable skb is from a network packet, thus, it is tainted. There is no bound check before a memcpy is used on skb -> data. Thus, an error will be generated because a tainted variable is doing memcpy.

Problem 6:

a. The process will have user id n because it is forked from a process with user id n.
b. (1) After the call the new process can have the user id m because before the call, the saved user id is m, and the real user id is m, and since the euid is not 0, the setuid can only set the euid to suid or ruid.
   (2) When n = 0, then it have root permission, it can set the euid to any id it wants because it has the highest permission.
c. The advantage of assigning separate uids allows the system to give each modules different permission so that each modules will have least privilege. This follows the least privilege principle so that it will make the program more save.
d. The 0 uid means the root permission. However, we want to limit the privilege of each module to be minimal. Thus, we don't want to give each module the root permission, we want to follow the least privilege principle to give each module the minimal permission to run in order to be safe.
e. Since a process with root permission to fork, after the fork, the new process will have root permission, too. Thus, usually setuid is called to set the new process's id to some nonzero number, so that the permission can be as least as possible in order to be safe. This also follows the least privileges principle to be safe.
f. (1) Since the password file is public readable and only can be written by processes with root privileges, the program which change this file has to run as root. To do this, we set the setuid bit of the passwd program to be 1, so that the program will be run as root, and it will be able to write the password file in order to change the password.
   Although the user's euid is not 0, but when the setuid bit is 1, the program will run as its own, so that the program will run as root, so that it can write the password file.
   (2) Because the passwd program will be run as root, if the program is being attacked, it can write anything (something terrible) to the password file since it has root permission. Thus, the passwd program has to be written very carefully because it can't be exploited because it has root permission and an attacker who got the root permission can basically do anything to the system.