1 - 6.7

site note:
Once you have connected a USB stick you found with malicious code in it to your work computer, you have already lost, and more likely so has your company.
This is alarmingly a serious issue as the Dept. Homeland Security ran a test by dropping USB sticks in gov building parking lots, and 60-90 % of the workers who picked them up actually plugged them into their work computers in those gov buildings.

Once the USB has been plugged in, if the malware was designed as most malware are, to get on as many systems as possible to reap the most benefits, then the wave of propagation mechanisms will take place instantly when the USB is plugged in. Immediately, the malware will find avenues for ways to replicate itself on other machines.

It will scan for other devices connected to the computer: other storage devices
It will look for email programs so that it my attach itself to an email in hopes that the recipient will download the attachment
It will look for programs that connect to remote systems/computers/networks, use the credentials on the host machine to connect to those remote machines to copy itself there and further propogate

In order to test a found USB stick, connect it to a computer with no networking hardware whatsoever. includes ethernet/wifi/Bluetooth (Though in order to fully understand what is in the USB, have 'fake' networking hardware/ software that mimics network ports, to see if any outgoing connections not made by the user are attempted). Basically an air-gaped computer is the only computer you should use to inspect the USB.

2 - 6.10

To be able to send SMS messages would allow the app to send maliciously crafted messages that could invoke downloading the app or other malware onto the recipients phone.
An address book contains email addresses on top of phone numbers, so the app would craft malicious emails to send to those email addresses.
The question stated it was a game, so perhaps you should look into the game online and see why it would require permission to send SMS messages.
The malware could be any type of malware, but based on wanting to access my address book and gain ability to send sms messages, it would be spyware/spear-fishing. A phishing attack success becomes higher when you have a bigger audience, thus aided by the malware attempting to duplicate itself over through the address book and sms.

It could also be a botnet style malware, as again, it wants to be on as many machines as possible, so its trying to send itself to as many machines as possible with the permssions its asking for.

<u>3</u>

Looking for specific parts of a virus would aid in detection of a virus in healthy code. There is always code that changes, but overall has the same effect and will always be doing the same thing. So if a virus infects a program, and that program infects another program, in that second program's healthy code, you'd be able look for instances you've seen the virus activate from the infection in the first program.

Because polymorphic viruses have the advantage of being the same virus but changing it's code around so that there are multiple signatures for the same virus which aids in obfuscation of the virus.
Scanning for module 'A' instead of all the modules is extremely helpful because you're never going to see the same signature for 'ABCD'; there would be too many changes the next time the virus replicates itself. It becomes easier to analyze a module, and search for instances of what that module is known to do, even if a new replication of the module in a replicated virus changes things like using different registers for doing things. You should at least know what the objective of 'A', and be able to scan for signature that is similar to what you know about A.

<u>4</u>

Encrypting control flow addresses like return addresses would prevent buffer overflow attacks because the attacker then doesn't know what the real return address is, and would prevent an attacker from overwriting the control flow data. However, the method of using the memory location of where the encrypted return address is stored as the key is where this protocol fails. An attacker can simply probe for where the program wants to return from the stack and jump somewhere, and the attacker would see where in memory the program wants to do that, and simply use that location in memory as the key for encryption.
It would be better to use a one time canary as the encryption, and save that canary elsewhere in the system as the encryption key where the system knows to look for that canary.

<u>5</u> - 10.11

the use of ***sprintf*** vs ***snprintf*** is analogous to the use of ***gets*** vs ***fgets***

sprint is dangerous because the output can be more characters than are allowed in the allocated size of a string.

in the function ***display(….)***, ***sprintf*** is used rather than ***snprintf***
simply replace the **sprintf** line in the **display** function with

```
void display(char *val)
{
 char tmp[16];
sprintf (tmp, 16,  "read val: %s\n", val);
puts(tmp);
}
```

the number 16 is the specified size of the maximum number of characters to output

<u>**6**</u> - 10.11

The function reads from stdin, and there is no checking or limitining of the amount of data that comes from stdin, so anything more than 63 chars will cause a buffer overflow. It wouldn't cause one if the program checks that no more than 63 chars will be accepted. fgets solves this problems.

So, in the main function @ **bolded-line**

```
……
prinft("Enter value:  ");
gets(next->inp);
next->process(next->inp);
……..
```

the bolded line should thus be: **fgets(next->inp, sizeof(next->inp), stdin);**

<u>**7/8**</u> – lab

<u>**TASK 1**</u>

In this lab, we look at a vulnerability in stack.c and must modify exploit.c in order to craft a file that would buffer overflow in stack.c and get you into a root shell.

stack.c uses ***strcpy***, which does not do bounds checking, and thus allows a buffer overflow

To do this, I compiled stack with the options mentioned in pdf as well as –g for gdb debugging to find the address in memory that stores the return address

made a breakpoint at the ***bof*** function
run bof – found address 0xbfe42f37
print buffer (***buffer*** in ***bof*** function) - gives address 0xbfe42ef8

Thus the return address should be 0xbfe42ef8, which is the end of the stack from for the ***buffer[24]***.
I was not able to get a root shell for Task 1, with the program only crashing and returning a segmentation fault.

TASK 2

With address randomization turned on, I would probably not be able to get a root shell. There would be no natural easy way of figuring out where in memory I would have to break the stack.
You would only get a segmentation fault, as I am getting, because now there is a random address where the program starts at.
We can brute force this though, by coming close to the actual address, and running a loop so that eventually you can break the program and get root shell.
Again, I was not able to achieve getting a root shell with a loop.

TASK 3

With Stack Guard left on, exploiting stack.c to get root shell becomes pretty much impossible. The program still gives segmentation fault/ program crashes, and root shell is still not obtained.
Turning on Stack Guard is basically an alarm system, so that when we overwrite past the end of the buffer, we over write a canary, which halts the program/crashes and terminates.

TASK 4

Making the stack non-executable still results in the program crashing with a segmentation fault. This protection prevents executable code like our shellcode from running in the stack we are trying to smash, so the program terminates/crashes, and so I cannot get a root shell.