Q1)

Interception: if the registration forms are not guarded (assuming they are on pieces of paper), anyone can grab them, thus compromising the confidentiality of all the participants

Interruption: someone changes all the bidding ID's on the registration forms, thus the auction cannot provide its service since they won't know who to send the auction items to.

Modification: the malicious person switches all the auction item numbers, so people end up bidding on the incorrect item, resulting in a loss of integrity for that organization as it appears as though they sent the wrong item.

Fabrication: someone brings their own auction item and replaces a real piece with theirs. When the bidding is completed the fake item is sent to the winner, while the person who replaced the piece has the original.

Online Auction:

Interception – someone could be reading the data of the users as they connect to the server.

Interruption – launch a ddos attack on the website until either the website crashes are load times are too long that users become uninterested.

Modification – when an email is sent to the winner, the email is modified to link to the hackers account or the instructions are changed.

Fabrication – create an email with the same template that a winning bidder would get and send it off to all participants.

A) Confidentiality
B) Availability/integrity
C) Privacy

Q2)

Prevent – getting rid of the passports altogether would prevent scammers from creating fake passports/cards completely as there would no longer be a market for them.

Deter – Adding a unique identifier as well (one which businesses could search and find your data to see if it matches the card). Unique identifiers would require figuring out the algorithm, which is quite a bit of work.

Deflect – creating a fine against fake vaccine users and creators, if they are to be caught the legal and financial implications may be very massive in size that faking a vaccine doesn't become worth the risk and the amount of work to replicate also becomes harder. This would make some scammers less inclined to make cards as the fear of being caught outweighs the profit.

Detect – continually checking for temperature, checking if there is extra information (ex: medical provider's name, reference number), fully printed cards. Seeing a massive spike in vaccinated cards/people when official vaccination numbers differ.

Recover: through these fake vaccinations, many people will begin to question the integrity of the system, in which case to recover from such an event the government must create a more secure way to implement proof of vaccination, for example unique QR codes for each user which is printed on your passport specifically after downloading from the official government website. Although expensive, another way to recover from this event would be to deprecate the current system and start anew.

Sploit2.c – class A (memory vulnerability)

This exploits the stack via a char buffer overflow (class A). The vulnerability exists within the while loop starting on line 37 and ending on line 41.

Based on the code, we see that the buffer is accessed via an integer variable 'i', however 'i' is never bound check and can therefore surpass the max size of the buffer array. Since we can write beyond the array, once we find the address of the return address we can change it to point to the beginning of the buffer, in which we have injected our shellcode.

Steps:

1) Identify the memory address that the return address is saved at
    - By knowing the memory address, we will know at what value of i we should begin to replace the return address with the address of the buffer
2) Identify the memory address that the buffer begins at and save this number in our code
    - In our buffer, we repeatedly store the address so that it matches the size interval that memory increments by
3) Inject shellcode at the beginning of the buffer

    - In our code, we create a replica buffer that spans from the beginning of the program buffer to the end of the return address, we place our shellcode at the beginning of the buffer and place the buffer address on repeat for the rest of the buffer
4) Since the stack grows upwards, we must be careful about changing the value ‘i' while reading past the buffer. To prevent the variable ‘i' from changing, we found the memory address on the stack, found the offset from the return address, and at the specific location changed the address in the buffer to mimic the value stored at ‘i' during the execution in which the buffer overflow would change i, this way it would remain unchanged and we could reach the return address
5) Finally when the program runs, we pass in the parameters, overwrite the return address and our shellcode is executed.

Fix:

The vulnerability can be fixed by having a check on the value of ‘i’ and ensuring it doesn’t go past the size of the buffer. Once i = buffer_size, stop the while loop (regardless if there is more to read or not). Essentially, add bounds checking.

Sploit3.c (non-memory vulnerability)

This exploit exploits a TOCTTOU vulnerability found in the program at lines 199 to 243.

The first check that is done (Time of Check) at line 199, where ‘realpath’ is executed on the provided path. From then on, no further checks on the file is done up until line 243 where copyFile is executed (TOU). In fact, due to the verification where a key is asked to be pressed to verify, there is a large window of time for a hacker to switch the file they originally passed in to be something else.

Steps:

1. Create a passwdfile containing an user that has root privileges but under another alias (say hacker) → hacker::0:0:root:/root:/bin/bash. This user “hacker”, with a UID of 0 has root privileges and no password
2. Run the backup file to create a backup of this new password file
3. Restore our backup from the backup directory, however once we reach the point of execution to verify the requested operation, our child process will create a symbolic link to /etc/passwd from our current psswd, and then we can verify the requested operation.
4. Since at this point no checks on the file is done, once the key is pressed, the program will continue along with our file now acting as a shortcut to /etc/password.

5. The restored file will write over the current passwd file location, and since we have created a symbolic link, /etc/password gets changed instead.
6. Since we have populated /etc/password with ‘hacker’, we can call system(“su hacker”) to login to hacker, who now has root privileges.

Fix:

- Use the file right after it has been authenticated, as in the next line after, so that the window given to the hacker has decreased to microseconds or even smaller, making it nearly impossible to create a symbolic link in time and therefore almost impossible to exploit.