# Software & System Security

**IMPORTANT NOTES: Study lecture materials at least 1 hour and prepare Question 1-4 prior to the tutorial session. Prepared questions will be discussed in the tutorial session.**

1. What is a *buffer overflow* and why can it be potentially used to attack a system? Explain in terms of abstract concepts (Input, Storage, Memory content, Memory addresses, etc.). What needs to be done at development time to prevent buffer overflow attacks? What can be done at run-time (or compile time).

   - Memory contains program code and data. If input is written into memory and is bigger than the space reserved for the data, it might overwrite other memory content, in particular memory content relevant for program execution, such as return addresses. Then, the program execution can change and the attacker can let the process execute commands that were not in the original program.
   - At development time, a program needs to include sufficient checks for input sanitization in order to prevent faulty (long) input to overwrite out of bounds memory.
   - At run-time (i.e. compile time) return addresses can be protected by so-called canaries that show that the particular memory segment was not changed. Also address randomization can make it much more difficult to jump to parts of the program or library that holds a function useful for the attack.



2. Give an example of a violation of the principle of least privilege, and explain how the system design could be modified to fix the violation.

   Consider the following example: A web application for managing sales of a large online merchant has several modules, two of them are: (1) Product catalogue management module: can add/remove/update details on database table CATALOGUE for products available for sale by the merchant, (2) Customer management module: can store personal details of customers and their purchases in a database table CUSTOMERS.





   Violation of principle of least privilege, Both modules (1) and (2) have root permission to the company database and can read/write to all tables stored including CUSTOMER and CATALOGUE. There is no need for those software modules to have root access. Module (1) needs read/write access to CATALOGUE table but not read/write access to the CUSTOMER table, Module (2) needs read/write access to CUSTOMER table but only read access to the CATALOGUE table. To fix it, both modules should be rewritten to reduce their privileges to the minimum required and not more.

3. What is a command injection vulnerability? Give an example and explain how to defend against such vulnerabilities.

   Command injection vulnerabilities occur when an attacker is able to input and execute an unauthorised command to a system. Usually such vulnerabilities occur due to lack of proper input validation or santisation, causing suitably formed user input data to be interpreted as a command. An example are programs forwarding user input (e.g. file names) to an underlying operating system, without any validation, allowing the attacker to inject OS commands to the system. A defense is to use sound input validation, and preferably to use APIs to the underlying system in which data and commands are provided as separate arguments to ensure that user data input cannot be mis-interpreted as a command.

4. Pick two of the following software design principles, and for each one, explain what it means, give an example of what could go wrong if the principle is not followed, and explain which security goal(s) (confidentiality, integrity, authenticity and availability) would be breached (It can be more than one goal(s)).

(a) Use secure defaults

(b) Defense in depth

(c) Separation of privilege

(d) Assume external systems/entities are insecure

(e) Authorize after authentication

(f) Only receive control instructions from trusted sources

(a) Use secure defaults: Without any user intervention, the state of the system should minimise the potential points of attack. For example, in an operating system, if the default settings when the system is freshly installed is to allow all incoming external network connections, malicious network connections could be established and hijack the system, which could impact all three security goals. It should by default disable all such connections. System should not allow any users to gain access and/or disclose any information when it defaults. The overall system should not be exposed unprotected.

(b) Defense in depth: Avoid single-points of failure; more than one mechanism/control should be broken in order to compromise the system's security. For example, if password-only authentication to a system is being used by a user, any loss of password by that user could result in full compromise of the user's authenticity. Instead, using two-factor authentication instead of one-factor authentication for multiple layers of validations for the identity can help in defending in depth. Single failure of one factor of validation will not leave the system unprotected.



(c) Separation of privilege: Separate the privileges of a user/process from other users/processes, to ensure that even if user/process A is compromised by an attacker, it will not affect the security of other users/processes. For example, if user Alice only needs read/write access to a her directory AliceDir in a file system but is given privileges of the system administrator to access all directories, then compromise of Alice's account by an attecker will give the attacker administrator priviliges to the full system. This will compromise the integrity of the whole system, rather than just Alice's directory AliceDir.





(d) Assume external systems/entities are insecure: Any external systems/entities not under full control of the system designer should be regarded as compromised by an attacker. For example, if a software system A processing sensitive data makes use of a cloud storage software service, it should not rely on that software service to protect the confidentiality of the sensitive data; if it passes unprotected sensitive data on the cloud server, that data could be exposed by the cloud service (or bugs in that service's software). This could for example affect the confidentality of system A's sensitive data. Instead, system A should protect the confidentiality of the data using encryption (discussed at a later lecture) before storing it on the untrusted cloud service.

(e) Authorize after authentication: A system should only authorise access to data by a user after the user has been authenticated, so that the appropriate access rights for the user can be granted (or denied). For example, a file system should allow access to a sensitive file only after authenticating the user and checking that user is allowed to access such sensitive data. Otherwise, a confidentiality or integrity breach could occur.

(f) Only receive control instructions from trusted sources: Only allow authorised parties to control a system. For example, a system that allows any user to inject commands via user data input (e.g. in OS command injectin attacks) can control the system regardless of the level of trust in the user. This can affect authenticity and integrity of a system if an untrusted party gains access to the system.

Other software design principles are also crucial when developing an application:

- *Keep design small and simple*: small and simple designs are easier to perform a security review and analysis on. Complex systems are more likely to have hidden functionality/security vulnerabilities that have been overlooked by security reviews.
- *Open design*: The design of a security mechanism should be open rather than secret. Open algorithms gains higher confidence as they are continuously being reviewed by experts. Historical experience shows that secret designs are sooner or later reverse-engineered by attackers, so the system security should not depend on the secrecy of the design and the algorithms used.

5. Consider the following C program. What type of software vulnerability is present in it, and how could it potentially be exploited by an attacker?

```
1:      #include <stdio.h>
2:      #include <string.h>

3:      int main(int argc, char *argv[]){

4:              char buffer[96];
5:              unsigned short s;
6:              int i;

7:              if(argc < 3){
8:                      return 1; /* error if less than 2 arguments */
9:                                /* (3 arguments including prog. name) */
10:             }

11:             i = atoi(argv[1]); /* convert first arg string to an int i */
12:             s = i;             /* convert int i to a short s */

13:             if(s >= 96){            /* (a) */
14:                     return -1;
15:             }

16:             printf("s = %d\n", s);

17:             memcpy(buffer, argv[2], i); /* (b) */
18:             buf[i] = '\0';
19:             printf("%s\n", buffer);

20:             return 0;
21:     }
```



Integer overflow is exploited to cause a buffer overflow. In C, int variables like i are represented by 32 bits and unsigned short integers like s, are represented by 16 bits. Variable s can hold $2^{16} = 65536$ values ranging from 0 to 65536-1. When the attacker inputs 65536 as the first argument after the program name (argv[1]), the value of i assigned in line 11 is 65536 which is within the range of values $-2^{31}, \ldots, 2^{31} - 1$ for an int. However, in the assignment statement on line 12, there is an integer overflow, since i=65536 = $2^{16}$ exceeds the maximal value $MAX = 2^{16} - 1$ for the short s. Therefore, the value assigned to s in line 12 will be s = i mod$MAX$ + 1 = 65536 mod 65536 = 0. i.e. s = 0. s is then used for comparing with 96, the max limit of buffer. In this case, s = 0 is smaller than 96 so that the overflow detection condition (s>= 96) is not fulfilled (and hence the overflow problem is undetected). The program continues.
Meanwhile, int variable i still has the value 65536 is stored in it. In (b), buffer flow occurs when copying argv[2] with size of 65536 bytes (greater than the 96 byte size of the buffer) is copied to buffer in memcpy() function.

To mitigate this issue, we should use i for comparison in line 13 ((a)) instead of s. And, an additional condition should be added to the line 13 (a) test as follows:

```
13:     if (i<=0 || i >= 96)    /* (a) */
```

The additional error condition check ($i <= 0$) to avoid the following alternative attack. If an attacker enters a negative number for i, it will be interepreted as a large positive number by memcpy (since memcpy interprets its number of bytes to copy argument as an *unsigned* int), whereas the comparison of (a) would treat it as a (signed) int and it will pass the test (since a negative number is smaller than 96), so an overflow could still occur in line 17. An alternative would be to define *i* as an unsigned int, rather than an int.