# Secure Software Design, 3rd CS – Dr. Reem Jafar Ismail 2011-2012

**Q1:** *Software security* is concerned with protection of software and its resources. A security risk is the probability of sustaining a loss of a specific magnitude during a specific time period due to a failure of security system.

a. ***Attackers' Knowledge***:- As software security continue to gain the attention of security experts and organizations, the tools and techniques used by attackers are becoming increasingly sophisticated and invasive.
b. ***Complexity of software***:- Software development is a complex process because it involves many activities and specialties. It involves different units, stages and personalities, and all these have to be integrated to achieve successful software production. The situation is even worse with the connection to Internet and the use of mobile codes.
c. ***Security Education***:- Producing secure software requires a lot of security training, education and experience. Many software developers are not well grounded in computer security. To what extent of the techniques and tools of attackers do they understand? Also, many programming books do not teach how to write secure programs. Many curricular of schools offering computer science do not properly address computer security.
d. ***Attitude of Software Engineers***:- In the past, software developers were only interested in producing software product with the desired quality. Little attention was paid on making secure products. But, the situation is changing now that everybody is aware of the implications of vulnerabilities in software.
e. ***Inadequacy of Computer Security Models***:- Many security models are still not adequate for producing secure software products. Implementing some of them is complex and difficult to realize.

**Q2:A:**

Application security can be defined as:

*"Application security is about protecting software and the system that software runs in a post-facto way, after development is complete*"

There is a considerable difference between the idea of software security and application security. As mentioned earlier, software security is about building/engineering secure software by planting security into the development lifecycle. It involves incorporating security into every phase of the development life cycle and educating stakeholders about the notion of security. On the other hand application security also means securing software after the development is completed, i.e. the securing software in a post facto way. In addition it also follows approaches such as penetrate and patch, and input filtering. Application security tends to secure software from known and exploited vulnerabilities. It does nothing towards building secure software. It is hard to secure defected software than defect-free software, for example, software having buffer overflow vulnerabilities can be protected by input filtering but a better technique would be to have software without buffer overflow vulnerabilities. Therefore in comparison with application security, it can be said that software security is a better solution than application security towards securing software.

**Q2: B:**

**Closed Source:**

Under the closed source model source code is not released to the public. Closed source software is maintained by a team who produces their product in a compiled executable state, which is what the market is allowed access to. Microsoft, the owner and developer of Windows and Microsoft Office, along with other major software companies, have long been proponents of this business model. Although in August 2010, Microsoft interoperability general manager Jean Paoli said Microsoft "loves open source" and its anti-open source position was a mistake.

**Open Source:**

Open source software refers to applications developed in which the user can access and alter the "source" code itself. Some open source applications have restrictions on their use and distribution, but many do not. Unlike public domain software, Open source software does have copyrights. Examples of open source software include: LINUX, Apache, Firefox, KOffice, Thunderbird, OpenOffice, and SquirrelMail.

**Q3: The student can answer one of the following approaches to "Validate Input Data"**

1. **Centralize the Approach**

Make your input validation strategy a core element of your application design. Consider a centralized approach to validation, for example, by using common validation and filtering code in shared libraries. This ensures that validation rules are applied consistently. It also reduces development effort and helps with future maintenance.

In many cases, individual fields require specific validation, for example, with specifically developed regular expressions. However, you can frequently factor out common routines to validate regularly used fields such as e-mail addresses, titles, names, postal addresses including ZIP or postal codes, and so on.

2. **Do Not Rely on Client-Side Validation**

Server-side code should perform its own validation. What if an attacker bypasses your client, or shuts off your client-side script routines, for example, by disabling JavaScript? Use client-side validation to help reduce the number of round trips to the server but do not rely on it for security. This is an example of defense in depth.

3. **Be Careful with Canonicalization Issues**

Data in canonical form is in its most standard or simplest form. Canonicalization is the process of converting data to its canonical form. File paths and URLs are particularly prone to canonicalization issues and many well-known exploits are a direct result of canonicalization bugs. For example, consider the following string that contains a file and path in its canonical form.

```
c:\temp\somefile.dat
```

The following strings could also represent the same file.

```
somefile.dat
c:\temp\subdir\..\somefile.dat
c:\  temp\   somefile.dat
..\somefile.dat
c%3A%5Ctemp%5Csubdir%5C%2E%2E%5Csomefile.dat
```

In the last example, characters have been specified in hexadecimal form:

```
%3A is the colon character.
%5C is the backslash character.
%2E is the dot character.
```

You should generally try to avoid designing applications that accept input file names from the user to avoid canonicalization issues. Consider alternative designs instead. For example, let the application determine the file name for the user.
If you do need to accept input file names, make sure they are strictly formed before making security decisions such as granting or denying access to the specified file.

### 4. Constrain, Reject, and Sanitize Your Input

The preferred approach to validating input is to constrain what you allow from the beginning. It is much easier to validate data for known valid types, patterns, and ranges than it is to validate data by looking for known bad characters. When you design your application, you know what your application expects. The range of valid data is generally a more finite set than potentially malicious input. However, for defense in depth you may also want to reject known bad input and then sanitize the input.
To create an effective input validation strategy, be aware of the following approaches and their tradeoffs:

- Constrain input.
- Validate data for type, length, format, and range.
- Reject known bad input.
- Sanitize input.

- **Constrain Input**

Constraining input is about allowing good data. This is the preferred approach. The idea here is to define a filter of acceptable input by using type, length, format, and range. Define what is acceptable input for your application fields and enforce it. Reject everything else as bad data.
Constraining input may involve setting character sets on the server so that you can establish the canonical form of the input in a localized way.

- **Validate Data for Type, Length, Format, and Range**

Use strong type checking on input data wherever possible, for example, in the classes used to manipulate and process the input data and in data access routines. For example, use parameterized stored procedures for data access to benefit from strong type checking of input fields.

String fields should also be length checked and in many cases checked for appropriate format. For example, ZIP codes, personal identification numbers, and so on have well defined formats that can be validated using regular expressions. Thorough checking is not only good programming practice; it makes it more difficult for an attacker to exploit your code. The attacker may get through your type check, but the length check may make executing his favorite attack more difficult.

- **Reject Known Bad Input**

Deny "bad" data; although do not rely completely on this approach. This approach is generally less effective than using the "allow" approach described earlier and it is best used in combination. To deny bad data assumes your application knows all the variations of malicious input. Remember that there are multiple ways to represent characters. This is another reason why "allow" is the preferred approach.

While useful for applications that are already deployed and when you cannot afford to make significant changes, the "deny" approach is not as robust as the "allow" approach because bad data, such as patterns that can be used to identify common attacks, do not remain constant. Valid data remains constant while the range of bad data may change over time.

- **Sanitize Input**

Sanitizing is about making potentially malicious data safe. It can be helpful when the range of input that is allowed cannot guarantee that the input is safe. This includes anything from stripping a null from the end of a user-supplied string to escaping out values so they are treated as literals.

Another common example of sanitizing input in Web applications is using URL encoding or HTML encoding to wrap data and treat it as literal text rather than executable script. HtmlEncode methods escape out HTML characters, and UrlEncode methods encode a URL so that it is a valid URI request.

5. **Encrypt sensitive cookie state**

Cookies may contain sensitive data such as session identifiers or data that is used as part of the server-side authorization process. To protect this type of data from unauthorized manipulation, use cryptography to encrypt the contents of the cookie. Make sure that users do not bypass your checks. Make sure that users do not bypass your checks by manipulating parameters. URL parameters can be manipulated by end users through the browser address text box. For example, the URL http://www/.<YourSite>/<YourApp>/sessionId=10 has a value of 10 that can be changed to some random number to receive different output. Make sure that you check this in server-side code, not in client-side JavaScript, which can be disabled in the browser.

6. **Validate all values sent from the client**

Restrict the fields that the user can enter and modify and validate all values coming from the client. If you have predefined values in your form fields, users can change them and post them back to receive different results. Permit only known good values wherever possible. For example, if the input field is for a state, only inputs matching a state postal code should be permitted.

7. **Do not trust HTTP header information**

HTTP headers are sent at the start of HTTP requests and HTTP responses. Your Web application should make sure it does not base any security decision on information in the HTTP headers because it is easy for an attacker to manipulate the header. For example, the referer field in the header contains the URL of the Web page from where the request originated. Do not make any security decisions based on the value of the referer field, for example, to check whether the request originated from a page generated by the Web application, because the field is easily falsified.

**Q4:** Buffer overflows begins with something every program needs, someplace to put stuff. Most useful computer programs create sections in memory for information storage, when continuous chunks of the same data type are allocated, the memory region is known as a **buffer**. The C programming language allows programmers to create storage at runtime in two different sections of memory: the stack and the heap.

When a buffer overflow happen, the excess data may trample other meaningful data that the program may wish to access in the future. Sometimes changing these other data can lead to a security problem.

In the simplest case, consider a Boolean flag allocated in memory directly after the buffer. Say that the flag determines whether the user running the program can access private files. If a malicious user can overwrite the buffer, then the value of the flag can be changed, thus providing the attacker with illegal access to private files.

**Q5:** Code obfuscation is to transform the code in such a way that it becomes more difficult for the attackers to read and understand. One common form of obfuscation is to rename all the variables in your code to arbitrary names. This obfuscation is not very effective though, so there are several simple techniques that can make the code more difficult to comprehend:

**1. Add code that never executes, or that does nothing.** if you add code that never executes, you need to keep it from being obvious that is never executes. One thing to do is to take calculations and make them far more complex than they need to be. Another thing to do is to use mathematical identities or other special information to construct conditions to loop conditions that always evaluate to either true or false. The idea is that a person or program deobfuscation your code will not be able to figure out that the condition should always evaluate the same way.

**2. Move code around.** Spread related functions as far apart as possible. Copy and rename the same functions, instead of calling it from multiple places. Combine multiple functions into single function that has some extra logic to make sure it calls the right block of code depending on how it got called. If an algorithm specifies that you do some operation A and then do B, move B as far from A as possible by putting other, unrelated tasks in between.

**3. Encode your data oddly.** Picking strings directly out of memory is easy if you don't take efforts to stop it. Convert everything to a strange character set, and only make string printable when necessary. Or, encrypt all you data in memory, using a set of keys that are spread throughout your program.

**Q6:**

- Access Control: Database access control deals with controlling who (a person or a certain computer program) is allowed to access what information in the database. The information may comprise specific database objects (e.g., record types, specific records, data structures), certain computations over certain objects (e.g., query types, or specific queries), or utilizing specific access paths to the former (e.g., using specific indexes or other data structures to access information).
  Database access controls are set by special authorized (by the database owner) personnel that uses dedicated protected security DBMS interfaces.
- Field Protection: Instead of relying on built-in-functionality of encryption, you should perform your own encryption and your own hashing at the application level using algorithms you trust. This is encrypt the data before storing it, and decrypts it on retrieval. You will likely need to take the binary output of the cipher and encode it to fit it into the database, because SQL commands do not accept arbitrary binary strings. We recommend encoding data using the base64 encoding algorithm, which does a good job of minimizing the size of the resulting data.

  One problem with self-encryption data in a database is that the database itself provides no way of searching such data efficiently. The only way to solve this problem is to retrieve all of the relevant data, and search through it after decryption.

  A bigger problem with encryption fields is that you generally end up expanding the data (because the fields generally must be stored as text and not the binary that results from an encryption operation). This may not be possible in some environments without expanding you tables. This is often prohibitive, especially if you've got a table design that has been highly tuned to match the underlying hardware or is something that can't be changed because it'll break many existing applications that expect a particular table layout.

**Q7:**

1. **Choosing a Language**

The single most important technology choice most software projects face is which programming language (or set of languages) to use for implementation. There are large number of factors that impact this choice. For example, efficiency is often a requirement, leading many projects to choose C and C++ as an implementation language.

Reliability is another factor that impact secure project, the more error checking a language can do the more reliable the programs written in that language. For that reason, Java has a distinct advantage because it has a much stronger static type system. Similarly, Java offers advantages over dynamic languages in which type errors and other mistakes only become apparent during runtime.

One of the biggest mistakes companies make in choosing a language is the failure to consider the impact on software security. However, many people either choose to ignore it completely or seem to assume that all languages are created equal when it comes to security. Unfortunately, this is not the case.

## 2. Choosing an Operating System

Modern operating systems are logically divided into a **system kernel** and user-level code (often called **user space**). Programs run in user space need to call down into the kernel when special services are needed.

Many critical services are run in kernel space. The kernel usually has some sort of security model that manage access to devices, files, processes, and objects. The underlying mechanism and the interface to that mechanism tend to be significantly different, depending on the operating system.

For programs running in user space, there are common security systems. One of the most important is **process space protection**. In a good operating system, a single process is not allowed to access any of the memory allocated to other processes directly. Additionally, no process can directly access the memory currently marked as “in use” by the operating system. In this way, interprocess communication is mediated by the operating system. Windows NT/2000 and all UNIX systems afford this kind of protection.

Windows 95/98/Me family of operating systems was not originally designed to afford the kind of protection modern operating systems provide. Operating systems like DOS were designed in a time when security was not a significant issue because most PCs were single-user devices.