CHAPTER 1
INTRODUCTION

1.1 ENTERPRISE WEB APPLICATION PARADIGM

Web application has become one of the most important communication channels between various kinds of service providers and clients. It has evolved from a static medium, with user interaction limited to navigation between web pages, to a highly interactive dynamic medium performing concurrent transactions and serving up personalized content. This dynamic medium application offers a wide range of services, such as on-line stores, e-commerce, social network services, etc. As more and more services are provided via the World Wide Web, efforts from both academia and industry are striving to create technologies and standards that meet the sophisticated requirements of today’s enterprise Web applications and users.

1.1.1 Web Application Architecture

A web application can be characterized by the number of layers that information will pass through on its journey from the data tier (where it is stored in a database typically) to the presentation tier (where it is displayed to the client). Each layer generally runs on a different system or in a different process space on the same system (Noack et al 2000).

A client – server architecture was adequate when data were primarily a text in the web applications. In the client – server system, the
client and server are separated by logics. This 2-tier client-server scenario works very well for a small business that only uses, or needs, a single data source (Lieven et al. 2005). However, the goal of most businesses is to grow, and this needs additional logics to meet the business demands. Unfortunately, the 2-tier approach does not scale very well. If the business rules change then the application needs to be rebuilt and redeployed.

Due to the limitations of the 2-tier client-server architecture, distributed applications are often divided into three or more tiers (Noack et al. 2000). Components in each of these perform a specific type of processing. There is a User Services (Presentation) tier, a Business Services tier, and a Data Services tier in a 3-tier application, as shown in Figure 1.1.

---

**Figure 1.1 Three-tier web applications architecture**

The 3-tier architecture, business logic, is separated from the user interface and the data source. Breaking up applications into these separate
tiers or sections can reduce the complexity of the overall application, and this results in applications that can meet the growing needs of today's businesses.

1.1.2 Web Applications

The structure of the Web application is the integration of a Hyper Text Markup Language (HTML), a Client Script and a Server script that will be formatted and distributed from web servers to client systems and viewed through a web browser. Web applications are placed in an appropriate place in the web server which will receive document requests and data submission from web browsers through the Hyper Text Transfer Protocol (HTTP) top of the Transfer Control Protocol / Internet Protocol (TCP/IP) layer (Andrews 2006). The main function of the web server is to feed HTML files to the web browsers. If the client requests an existing static file, it will be retrieved from the server’s hard disk and sent back to the web browser. The communication between the web browser and the web server is generally stateless.

The complexity of web-based applications has also grown significantly from information dissemination to online transactions, enterprise-wide planning, scheduling systems and Web-based collaborative work environments. Several attributes of quality web-based applications such as ease of navigation, accessibility, scalability, maintainability, usability, compatibility, interoperability, security and reliability are not given the consideration they deserve during development. Many web applications also fail to address the cultural, privacy, moral, and legal aspects. In order to address these main issues, all the web applications must have web security features.
1.1.3 Web Application Components

A component is an encapsulated piece of code that performs a certain function for an application. This function could be the processing of a business rule, or it could be the retrieval of some information from a database for an application. Generally, User Interface components are at the presentation layer (Kadri et al 2007), business components are at the business layer and database related components are at the data access layer.

The data centric component or the data access layer is responsible for integrating with the data sources like Relational Database Management System (RDBMS) tools, that the application needs, to be able to function. These data sources could be the SQL Server or Oracle databases, Exchange message stores, or UNIX legacy applications. It provides consistent data access to different data sources, and makes the location of the data transparent. The business component or the business logic layer is designed to hide the complicated interactions that a set of business rules needs, in order to process and shield the user interface designer from having to know anything about the underlying data. The necessity to use the business component is to encapsulate business rules inside a component. The user interface component or the presentation layer presents the application content to the client which would be generated by the business logic. Hence, an abstraction in the business logic component, data component and user interface component can be provided in the web application security.

1.1.4 Web Functionality

Web applications employ numerous different technologies to deliver their functionality. Reasonable functional application may employ dozens of distinct technologies within its server and client components.
1.1.4.1 Server-Side Functionality

Whenever a user requests a resource, the server’s response is created on the fly, and each user may receive content that is uniquely customized for him. When a user’s browser makes a request for a dynamic resource, it does not normally ask for a copy of that resource. In general, it will also submit various parameters along with its request. These parameters enable the server-side application to generate content that is tailored to the individual user. There are three main ways in which HTTP requests can be used to send parameters to the application, namely, the URL query string, HTTP cookies and the body of requests using the GET / POST method.

In addition to these primary sources of input, the server-side application may, in principle, use any part of the HTTP request as an input to its processing. Web applications employ a wide range of technologies on the server side to deliver their functionality. Some of them are given below

- Scripting languages, such as PHP, VBScript, JavaScript and Perl.
- Web application platforms, such as ASP.NET and Java.
- Web servers, such as Apache, Tomcat, IIS, and Netscape Enterprise.
- Databases, such as MS-SQL, Oracle, and MySQL.
- Other back-end components, such as file systems, SOAP-based web services, and directory services.

All these technologies are to be properly integrated and configured to provide secure service to the web clients.
1.1.4.2 Client-side functionality

In order to make the server-side application receive user input and actions, and present the results of these back to the user, it needs to provide a client-side user interface. Since all web applications are accessed via the web browser, all these interfaces share a common core of technologies. HTML forms are the usual mechanism for allowing users to enter arbitrary input via the browser. Hyperlinks and form elements can be used to create a rich user interface capable of easily gathering most kinds of input which web applications require. However, most applications employ a more distributed model, in which the client side is used not simply to submit the user data and actions but also to perform the actual processing of data.

A significant development in the use of JavaScript has been the appearance of Asynchronous JavaScript and XML (AJAX) techniques for creating a smoother user experience, which is closer to that provided by the traditional desktop applications (Zepeda and Chapa 2007). AJAX involves issuing dynamic HTTP requests from within the HTML page, to exchange data with the server and update the current web page accordingly, without loading a new page altogether. These techniques can provide very rich and satisfying user interfaces.

The integration of various complex technologies and an inadequate design may introduce vulnerabilities, which attract attackers to create threats in web applications. Most of the vulnerabilities in web applications are due to the poor development of web applications that continue to expand a high probability of low performance and/or failure.
1.1.4.3 **State and Sessions in web applications**

To implement most kinds of useful functionality, however, applications need to track the state of each user’s interaction with the application across multiple requests. Information normally held within a server-side structure is called a session. In some applications, state information is stored on the client component rather than the server (Kolsek 2002). A set of data is passed to the client in each server response, and is sent back to the server in each client request. Since any data transmitted via the client component may be modified by the user, applications need to take measures to protect themselves from attackers, who may change this state information in an attempt to interfere with the application’s logic. Since the HTTP protocol is itself stateless, most applications need a means of re-identifying individual users across multiple requests, in order to correct a set of state data to be used to process each request. This is normally achieved by issuing a token to each user, which uniquely identifies that user’s session. These tokens may be transmitted using any type of request parameter, but HTTP cookies are used by most applications.

Several kinds of vulnerability arise in relation to session handling. Attackers want to know the value of the session cookie. If the attacker succeeds, then he can hijack the victim’s session and compromise the user’s account and any sensitive data it holds. The attacker does this by inserting the stolen session cookie into its browser and making requests to the application. Since authentication is based solely on the cookie, and the HTTP is a stateless protocol, the server is no longer able to tell the difference between the attacker and the victim (Palmer 2008). In addition to session hijacking, the other major security issues in web applications are the validation of the input and output data, Direct data access by compromising the vulnerabilities, Data
Poisoning, Malicious file execution, and poor system architecture and configuration (IBM White Paper 2008).

Hence, considering all aspects like different categories of enterprise web application systems, different web application components, various client and server technology functionalities, and web applications states and sessions, the security in web applications is absolutely necessary to protect the valuable assets from the unknown external user.

1.2 HOLISTIC WEB SECURITY

To understand web application security, the threats, the vulnerabilities, and attacks have to be defined. A threat is any potential malicious occurrence that could harm an asset. A vulnerability is a weakness that makes a threat possible. This may be because of poor design, configuration mistakes, or inappropriate and insecure coding techniques.

![Figure 1.2 Holistic security approach](image)
An attack is an action that exploits a vulnerability or enacts a threat. Examples of attacks include sending malicious input to an application or flooding a network in an attempt to deny service. Security on assets could be implemented at three levels in the Internet architecture, such as the Network level, Host level and Application level (Meier et al 2003), as shown in Figure 1.2.

Network Security is a basic form of security for Internet transmissions for blocking or filtering of data from unknown or suspect sources. One way to accomplish this is by restricting the number of open communication ports on the server, limiting inbound transmissions to those protocols supported by the few open ports. In a network, a firewall can help to prevent hackers or malicious software (such as worms) from gaining access to other computers through a network or the Internet. The difficulty for firewalls is distinguishing between legitimate and illegitimate traffic. If firewalls are configured correctly, they can be a reasonable form of protection from external threats, including some Denial of Service (DoS) attacks. Regardless of the approaches taken to identify and protect against attacks from outside the network, there remains a larger and perhaps more insidious threat that can occur from within the network itself. So, securing the host becomes an important task within an organization. Attacks may come from hostile employees, from persons with access to a particular location, or from many other sources. Hence, protecting the resources within the organization is a tough challenge in security. Patch management is a common solution to secure the host and it is an essential sub-process of release management within the overall software maintenance (Sihvonen and Jantti 2010). With a seemingly endless procession of software vulnerabilities and exploits, patching has become an important part of day-to-day network administration.
The Inner-level security refers to methods of protecting Web applications at the application layer, from malicious attacks that may expose private information. Generally, the web forms are vulnerable to application-level attacks. The main reason for this is that web designers implicitly trust validation rules which are enforced only on the client-side. In addition, application-layer attacks are very attractive to potential attackers, because the information they seek ultimately resides within the application itself, and it is easy for them to make an impact and reach their goals. Lower layer technologies that support application layer security such as Secure Sockets Layer (SSL) and the successor Transport Layer Security (TLS) are cryptographic protocols which provide secure communications on the Internet. The IPsec provides security services at the IP layer by enabling a system to select the required security protocols, to determine the algorithm(s) to be used for the service(s), and to put in place any cryptographic keys required to provide the requested services (Sierra 2002). But, Marsh Ray and Steve Dispensa discovered a design flaw in the TLS protocol that affects all versions of the protocol. Since, the vulnerability itself is serious, it need not affect many deployments once the administrators apply suitable patches to disable renegotiation, leaving TLS sufficiently secure in most cases, because exploiting the vulnerability requires the attacker to be an active man-in-the-middle, redirecting traffic between victims (Farrell 2010).

Hence, Web application security is the need of the hour for today’s web medium to provide secure and seamless services in an enterprise web application environment.

1.3 WEB APPLICATION SECURITY

Web applications are complex entities and designed to perform a specific function directly for the user or for another application program.
A web application includes a code that resides in the Web servers, application servers, databases, and backend systems of an organization.

![Diagram of web application attack model]

**Figure 1.3  Web application attack model**

Securing a Web application is very difficult, not only because of the cross-departmental coordination involved, but due to the fact that most security tools are not designed to address the web application as a whole, including how the different pieces of the application interact with each other. The potential for a security breach exists in each layer of a web application. The main reason for the threat would be the compromise in the vulnerabilities in web application, as shown in Figure 1.3. There are many reasons for security flaws to work their way into web applications, such as security is rarely considered during the functional requirements phase; in fact application owners do not even demand security from the onset, so that developers do not build security into their applications, and even when they do consider
security, they cover only the basics: authentication, authorization, access control and encryption. They often do not provide comprehensive input validation to prevent code injection vulnerable attacks, such as SQL-injection, XPath Injection, Cross-Site Scripting (XSS) and Session Hijacking.

### 1.3.1 Threat Classification Taxonomy

Threats faced by the web application can be categorized based on the goals and purposes of the attacks. As per the Web Application Security Consortium (WASC), threats are broadly classified (WASC 2010) as, threats in authentication, and authorization, client-side attacks, command execution threats, information disclosure threats, and logical attacks, as shown in Table 1.1. Threats in authentication target a web site and authentication is a method of validating the identity of a user. Authorization is a method of determining if a user, service, or application has the necessary permission to perform a requested action. Client-Side attacks that target vulnerabilities in client applications that interact with a malicious server or process malicious data. Here, the client initiates the connection that could result in an attack. The command execution attacks are designed to execute remote commands on the web site. All web sites utilize user-supplied input to fulfil requests. Often these user-supplied data are used to create construct commands resulting in a dynamic web page content. If this process is done insecurely, an attacker could alter the command execution. The information disclosure attacks are designed to acquire system specific information about a web site. System specific information includes the software distribution, versions’ numbers, and patch levels, or the information that contains the location of backup files and temporary files. The logical attacks focus on the abuse or exploitation of a web application’s logic flow. Application logic is the expected procedural flow used in order to perform a certain action.
<table>
<thead>
<tr>
<th>Threat Category</th>
<th>Type of Attack / Threat</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Based Attacks</td>
<td>Brute – Force Attack</td>
<td>A brute force attack is a method to determine an unknown value by using an automated process to try a large number of possible values.</td>
</tr>
<tr>
<td></td>
<td>Insufficient Authentication</td>
<td>Insufficient Authentication occurs when a web site permits an attacker to access sensitive content or functionality without having to properly authenticate.</td>
</tr>
<tr>
<td></td>
<td>Weak Password Recovery Validation</td>
<td>Insufficient password recovery is when a web site permits an attacker to illegally obtain, change or recover another user’s password. Password recovery systems may be compromised through the use of brute force attacks, inherent system weaknesses, or easily guessed secret questions.</td>
</tr>
<tr>
<td>Authorization based Attacks</td>
<td>Credential / Session Prediction</td>
<td>Credential/Session Prediction is a method of hijacking or impersonating a web site user. If an attacker is able to predict or guess the session ID of another user, fraudulent activity is possible.</td>
</tr>
<tr>
<td></td>
<td>Insufficient Authorization</td>
<td>Insufficient Authorization results when an application does not perform adequate authorization checks to ensure that the user is performing a function or accessing data in a manner consistent with the security policy.</td>
</tr>
<tr>
<td></td>
<td>Insufficient Session Expiration</td>
<td>Insufficient Session Expiration occurs when a Web application permits an attacker to reuse old session credentials or session IDs for authorization.</td>
</tr>
<tr>
<td></td>
<td>Session Fixation</td>
<td>Session Fixation is an attack technique that forces a user’s session ID to an explicit value.</td>
</tr>
<tr>
<td>Client Side Attacks</td>
<td>Content spoofing</td>
<td>Content Spoofing is an attack technique that allows an attacker to inject a malicious payload, that is later misrepresented as the legitimate content of a web application.</td>
</tr>
<tr>
<td>Threat Category</td>
<td>Type of Attack / Threat</td>
<td>Issues</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td>Cross-Site Scripting (XSS)</td>
<td>Cross-Site Scripting (XSS) is an attack technique that involves echoing an attacker-supplied code into a user’s browser instance.</td>
</tr>
<tr>
<td>Command Execution</td>
<td>Buffer Overflow</td>
<td>A Buffer Overflow is a flaw that occurs when more data is written to a block of memory, or buffer, than the buffer is allocated to hold.</td>
</tr>
<tr>
<td>Format String Attack</td>
<td>Format String Attack</td>
<td>Format String Attacks alter the flow of an application by using string formatting library features to access other memory space.</td>
</tr>
<tr>
<td>Light Weight Directory Access Protocol (LDAP) Injection</td>
<td>LDAP Injection</td>
<td>LDAP Injection is an attack technique used to exploit web sites that construct LDAP statements from user-supplied input.</td>
</tr>
<tr>
<td>OS Commanding</td>
<td>OS Commanding</td>
<td>OS Commanding is an attack technique used for unauthorized execution of operating system commands.</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>SQL Injection</td>
<td>SQL Injection is an attack technique used to exploit applications that construct SQL statements from user-supplied input. When successful, the attacker is able to change the logic of the SQL statements executed against the database.</td>
</tr>
<tr>
<td>SSI Injection</td>
<td>SSI Injection</td>
<td>SSI Injection (Server-side Include) is a server-side exploiting technique that allows an attacker to code into a web application, which will later be executed locally by the web server.</td>
</tr>
<tr>
<td>XPath Injection</td>
<td>XPath Injection</td>
<td>XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents.</td>
</tr>
<tr>
<td>Threat Category</td>
<td>Type of Attack / Threat</td>
<td>Issues</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Information Disclosure</td>
<td>Directory Indexing</td>
<td>Automatic directory listing/indexing is a web server function that lists all the files within a requested directory, if the normal base file is not present.</td>
</tr>
<tr>
<td>Information Leakage</td>
<td>Information Leakage</td>
<td>Information Leakage is an application weakness where an application reveals sensitive data, such as the technical details of the web application, environment, or user-specific data.</td>
</tr>
<tr>
<td>Path Traversal</td>
<td>Path Traversal</td>
<td>The Path Traversal attack technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory.</td>
</tr>
<tr>
<td>Predictable Resource Location</td>
<td>Predictable Resource Location</td>
<td>Predictable Resource Location is an attack technique used to uncover hidden web site content and functionality.</td>
</tr>
<tr>
<td>Logical Attacks</td>
<td>Abuse of Functionality</td>
<td>Abuse of Functionality is an attack technique that uses a web site’s own features and functionality to attack itself or others.</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Denial of Service</td>
<td>Denial of Service (DoS) is an attack technique with the intent of preventing a web site from serving normal user activity. DoS attacks, which are easily normally applied to the network layer, are also possible at the application layer</td>
</tr>
<tr>
<td>Insufficient Anti-Automation</td>
<td>Insufficient Anti-Automation</td>
<td>Insufficient Anti-automation occurs when a web application permits an attacker to automate a process that was originally designed to be performed only in a manual fashion</td>
</tr>
<tr>
<td>Insufficient Process Validation</td>
<td>Insufficient Process Validation</td>
<td>Insufficient Process Validation occurs when a web application fails to prevent an attacker from circumventing the intended flow or business logic of the application.</td>
</tr>
</tbody>
</table>
Application-level vulnerabilities, which are believed to account for 70% to 90% of the overall flaws, are now the main focus of attackers and researchers (Janot and Zavarsky 2008). Online applications (websites and services) are especially at high risk due to their universal exposure, and their extensive use of the firewall-friendly HTTP protocol. Moreover, database security is too often overlooked in favour of web and application server security, resulting in backend databases being a major target for attackers which are able to use them as easy entry points to the organizations’ networks.

Among the various types of application level web attacks, the two main attack techniques that have been used widely are the SQL Injection and Cross Site Scripting attacks. SQL Injection and Cross-Site Scripting are the prevalent types of attack in web applications (Fonseca et al 2009) due to the serious consequences, such as the logic to access, modify, and delete data in the database, possibly including the execution of system commands or creating a denial of service, and to control the appearance of a web site, transfer sensitive data, and hijack the session to take control of the user’s account. Though several effective prevention methods have been developed to protect against SQL Injections and Cross-Site scripting, these two attacks remain an issue on a practical level. XPath injection is very similar to SQL Injection. XPath injection can be possible where the XML data is being used as data logic instead of commercial RDBMS. Cross site scripting is further used to create session hijacking attacks through Session ID fixation, Browser hijacking and Background XSS propagation. So, SQL Injection, XPath injection, Cross-site scripting and Session hijacking are code injection vulnerabilities which leads to create threat to web applications (Johns 2011).
Hence, this thesis focuses on a novel design approach to prevent Code Injection Vulnerable attacks, such as SQL Injection, XPath injection, Cross-Site scripting attacks and Session Hijacking in web applications.

1.4 SQL INJECTION

Web applications in the World Wide Web (WWW) have user interfaces where a user can input data to interact with the application’s underlying relational database management system. This input becomes a part of an SQL statement, which is then executed on the RDBMS. A code injection attack that exploits the vulnerabilities of these interfaces is called an “SQL Injection Attack” (SQLIA) (CERT 2007; Su and Wassermann 2006; Howard and LeBlanc 2003; Viega and McGraw 2001).

Through SQL injection attacks, an attacker may extract undisclosed data, bypass authentication, escalate privileges, modify the content of the database, execute a denial-of-service attack, or execute remote commands to transfer and install software (Maor and Shulman 2005). SQL Injection could be possible through redirecting and reshaping the query, error based query, injection through stored procedure and blind SQL injection.

1.4.1 Redirecting and Reshaping the Query

The user inputs become an injected statement into the pre-written SQL Statement. By redirecting and reshaping the query, the attacker can enter data into a database, so that the web pages are changed to send a visitor to another page. Redirection and reshaping can be done through Tautology, End of Line Comment, Union Query, Piggy-backed Query and System Stored Procedure (Sun and Beznosov 2008).
1.4.1.1 Tautology

By using the tautology technique, the attacker intends to bypass authentication, identifying injectable parameters, and extracting data. The general goal of a tautology-based attack is to inject a code in one or more conditional statements, so that they always evaluate to true as shown in Figure 1.4. The consequences of this attack depend on how the results of the query are used within the application. In this type of injection, an attacker exploits an injectable field that is used in a query’s WHERE conditional operator (Khochare et al. 2011). Transforming the conditional into a tautology causes all the rows in the database table targeted by the query to be returned.

For Example, an attacker submits “’ or 1=1 - -” for the input field (the input submitted for the other fields is irrelevant). The resulting query is:

```
SELECT * FROM loginid, loginpwd from logintab where loginid='anything' or ‘1’=’1’
```

Figure 1.4 Tautology query – SQL injection
The code injected in the conditional (or 1=1) transforms the entire WHERE clause into a tautology. Since the condition is a tautology, the query evaluates to true for each row in the table and returns all of them.

1.4.1.2 Union query

In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query (Das et al 2011). With this technique, an attacker can trick the application into returning the data from a table, different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query> as shown in Figure 1.5. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query. For example, consider a query as shown in Figure 1.5; the first query leads to a tautology output and the second query returns the data from the “acctab” table. In this case, the database would return column “acctid” and “acctname” from the “acctab” which is an injected query along with the first query.

```
SELECT uid, uname FROM User WHERE uid='anything' or '1'='1'
UNION ALL SELECT accid, accname FROM acctab;
```

![Figure 1.5 Union query – SQL injection](Image)
The database takes the results of these two queries, unions them, and returns them to the application.

1.4.1.3 Piggy-backed query

The piggy-backed Query attack technique allows the attacker to extract data from the database, adding or modifying data, performing denial of service and executing remote commands (Kindy and Pathan 2011). In this attack type, an attacker tries to inject additional queries into the original query that “piggy-back” on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first as shown in Figure 1.6.

```
Select loginid, loginpwd from logintab where loginid='shan' and loginpwd='neethi';
Drop table emptab - -'
```

![Figure 1.6 Piggyback query – SQL injection](image)
After completing the first query, the database would recognize the query delimiter (";") and execute the injected second query. The result of executing the second query would be to drop table emptab, which would destroy valuable information.

1.4.2 Error Based Query

This technique is usually used by the attacker during the information gathering stage of the attack, which is considered as a preliminary method to gather information (Sun and Beznosov 2008). By injecting illegal/logically incorrect requests, an attacker may gain knowledge that aids the attack, such as finding out the injectable parameters, data types of columns within the tables and names of tables. The vulnerability leveraged by this attack is that the default error page returned by the application servers is often excessively descriptive. In fact, the generated error messages can often reveal vulnerable/injectable parameters to an attacker. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntactical errors can be used to identify injectable parameters. For example, the attacker injects the following text into the input field pin: "convert(int,(select top 1 name from sysobjects where xtype='u'))". The resulting query is:

```
SELECT accounts FROM users WHERE login=''' AND pass=''' AND pin= convert (int,(select top 1 name from sysobjects where xtype='u'))
```

In the attack string, the injected select query attempts to extract the first user table (xtype='u') from the database’s metadata table (assume the application is uses the Microsoft SQL Server, for which the metadata table is called sysobjects). The query then tries to convert this table name into an integer. Since this is not a legal type conversion, the database throws an error. For Microsoft,
There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first user-defined table in the database: “CreditCards.” A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information.

1.4.3 Injection through Stored Procedure

SQL Injection of this type tries to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database, and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQL Injection attacks can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. It is a common misconception that using stored procedures to write web applications renders them invulnerable to SQLIAs. Additionally, since stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run an arbitrary code on the server or escalate their privileges (Fayo 2005).
In the above example, assume that the query string constructed at lines 5, 6 and 7 has been replaced with the following statements.

```sql
SELECT accounts FROM users WHERE login='doe' AND pass=' ' SHUTDOWN; -- AND pin=
```

The stored procedure returns a true/false value to indicate whether the user’s credentials are authenticated correctly. To launch an SQLIA, the attacker simply injects “’ ; SHUTDOWN; --” into either the userName or password fields.

Good programming practices do not guarantee any foolproof defence to detect the vulnerabilities and eliminate them. Various other techniques like escaping the quotes and limiting the length of the user inputs are employed as a quick fix solution (Howard and LeBlanc 2003). It is of even greater concern that well known database vendor products like Microsoft SQL Server provide attackers direct access to the command line shell and registry, using methods like xp cmdshell and xp regread.

### 1.4.4 Blind SQL Injection

Blind SQL injection is identical to normal SQL Injection except that when an attacker attempts to exploit an application rather than getting a useful error message he gets a generic page specified by the developer instead (Evettev 2010). A blind SQL Injection vulnerability appears, when an attacker is not able to control the data shown to the user. The displayed data can be the
result of a vulnerable SQL request, when an injection gets into two different SELECT queries that, in turn, implement the selection from tables with different numbers of columns, and the filtering of query concatenation is used. The capabilities of blind SQL injection are comparable with those of the classical SQL Injection technique. Just like the classical technique of exploitation, the blind SQL Injection exploitation allows one to write and read files and get data from tables, only the entries are read symbol-by-symbol. Classical blind exploitation is based on the analysis of the true/false logical expression. If the expression is true, then the web application will return certain content, and if it is false, the application will return some other content. If an attacker considers the difference in the outputs for true and false statements in the query, he will be able to conduct symbol-by-symbol search for data in a table or a file.

1.5 XPath INJECTION

XPath is a standard language used to refer to parts of an eXtensible Markup Language (XML) document. It can be used directly by an application to query an XML document. Today, many organizations have adopted XML as a data format for everything from configuration files to remote procedure calls (Qiao et al 2009). So, like any other application or technology that allows outside user submission data, XML applications can be susceptible to code injection attacks, specifically XPath injection attacks. Moreover, its notation/syntax is always implementation independent, which means the attack may be automated. By sending intentionally malformed information into the web site, an attacker can find out the structure of the XML data, or the way to access data (Antunes et al 2009). An attacker may even be able to elevate the privileges on the web site if the XML data is being used for authentication and other transactions.
1.5.1 XPath Injection Motivation and Consequences

Website defacement which results in an unauthorized change to web applications is the top motivation for hackers. In XPath injection, when a malicious user may insert an arbitrary XPath code into the form fields and URL query parameters, in order to inject this code directly into the XPath query parser engine, doing so would allow a malicious user to bypass authentication (if an XML-based authentication system is used) or to access restricted data from the XML data source. This leads to privilege escalation and information leakage. Consider an application that uses an XML database to authenticate its users. The application retrieves the user id and password from a request, and forms an XPath expression to query the database. An attacker can successfully bypass authentication and login without valid credentials through XPath injection. Improper validation of the user-controlled input, and the use of a non-parameterized XPath expression enable the attacker to injection an XPath expression that causes the authentication to bypass. For example, consider an application where the user supplied the username and password, placed in the appropriate place of the XQuery to perform user authentication. The following XQuery is generated in the server and it would be sent to the xml document for user validation.

\[
let \text{Str} := \text{doc("login.xml")/Login/user}
\]\[
\text{return if } (\text{Str/username/text()='shan' and Str/passwd/text()='neethi'}) \text{ then } <b>true</b> \text{ else } <b>false</b>
\]

If the user name and password are presented in the XML data then this will return true to the web page; otherwise, it will return ‘false’. This is a simple authentication procedure in the web application, which uses the XML data as back-end service.
An attacker can craft the input in such a way that, the user always becomes an authenticated user for a web site by XPath injection. Hence, this XPath injection also leads to extracting the document structure and modifying the document information in addition to escalating the privileges.

1.6 CROSS-SITE SCRIPTING ATTACK

Web developers may use the Client Script, in particular, the JavaScript to enhance their web applications. If the attacker is able to embed a script in a web page that a valid user requests from sites, he can take control of the entire context. Such a maliciously embedded script is called a Cross-Site Scripting (XSS) attack (Kerschbaum 2007), since the script actually originated from a different site. In order to mount this attack, the attacker exploits the vulnerable web applications. Interactive web applications require a user input. If this input is used by the web application to build the response web page, it might be exploited by the attacker. In particular, if the inputs are not sanitized well enough by the application, then they can be exploited. Consider the following web application: A user requests a web page and the application responds with a personalized web page which is given below.

```

<html>
<body>
Hello, Neethi
...

If an attacker can craft the input with additional script in the requesting page

then the response might look this:

```html
<html>
<body>
Hello, Neethi
<script>alert("XSS Script")</script>
</body>
</html>
```

This script originating in the input part of the URL of the request would be executed in the attacked user’s security context with web site website.com. The consequences can be severe like Cookie (and session) theft, Browser hijacking, including but not limited to, User monitoring and data theft, and Request forgery and fake transactions. The different types of Cross-site Scripting attacks are persistent or Stored XSS, Non-Persistent or Reflected XSS and Document Object Model (DOM) based XSS.

### 1.6.1 Persistent XSS Attack

The persistent (or stored) XSS vulnerability is a more devastating variant of a cross-site scripting flaw. The persistent XSS vulnerabilities include all cases, in which the malicious input could be stored somewhere in the web application, waiting for a user to execute the script by requesting the relevant web page (Nadkarni et al 2011). Examples of cross-site scripting often include sites that allow a web application’s user to post messages or comments to be viewed by other users later. If the application stores the script and displays it to other users, that so the script is executed by the viewer’s browser, the cross-site scripting attack is successful.
Here is a PHP code that suffers from persistent XSS:

```php
<?php
if(isset($_POST['btnSign']))
{
$message=trim($_POST['mtxMessage']);
$name=trim($_POST['txtName']);
// Sanitize message input
$message = stripslashes($message);
$message = mysql_real_escape_string($message);
// Sanitize name input
$name = mysql_real_escape_string($name);
$query = "INSERT INTO guestbook (comment,name) VALUES ('$message','$name');";
$result=mysql_query($query) or die("<pre>'.mysql_error().'</pre>");
}
??>
```

In the above example, the two parameters in that code “message” and “name” are not sanitized properly. If an attacker enters malicious data for those inputs then those parameters will be stored in the guestbook table, which will be the pathway for the persistent XSS attack.

### 1.6.2 Non-Persistent XSS Attack

This type of XSS attack (and also referred to as the reflected XSS attack), exploits the vulnerability that appears in a web application when it utilizes the information provided by the user in order to generate an outgoing page for that user. In this category, instead of storing the malicious code embedded in a message by the attacker, the malicious code itself is directly reflected back to the user by means of a third party mechanism (Garcia-Alfaro and Navarro-Arribas 2007). By using a spoofed email, for instance, the attacker can trick the victim to click a link, which contains the malicious code. If so, that code is finally sent back to the user but from the trusted context of the application’s web site.
A non-persistent XSS attack is by far the commonest type of XSS attacks against current web applications, and it is usually combined together with other techniques such as phishing and social engineering (Jagatic et al 2007), in order to achieve its objectives. For example, if a search engine has a text box for a search keyword, then a user makes a search request such as

```
```

The above request will give the reply in connection with web application vulnerability. If the same input is modified as shown,

```
http://www.mysearchsite.com/index.html?searchword =<script>alert('You just found a XSS vulnerability')</script>
```

If this pops up an alert message box stating "You just found a XSS vulnerability", then this parameter is vulnerable to XSS attacks. This search engine text box will allow anything to be processed as a name, including a malicious script that is injected into the parameter passed in.

**1.6.3 DOM based XSS Attack**

While a traditional cross-site scripting vulnerability occurs on the server-side code, the Document Object Model based cross-site scripting is a type of vulnerability which affects the script code in the client’s browser. DOM is a way in which scripts can access the structure of a page in which they reside, and is used to manipulate the page content in WEB 2.0 applications. Like server-side scripts, client-side scripts can also accept user input, which can contain a malicious code. Therefore, if the client-side script inputs are not properly sanitized, they can be prone to DOM XSS
vulnerabilities. The possible sources of user inputs which can contain attack vectors, are ‘document.referrer’ property, ‘window.name’ property and ‘location’ property. When used without proper sanitization these user inputs can get into the code which is executed on the client-side, within the same context as the legitimate code from the server. The possible means by which an attack is executed are:

- `document.write`
- by changing the location with javascript: metaprotocol
- by eval, setInterval or setTimeout functions

For example, the ‘document.referrer ’ property is set by the browser and represents the page which is linked to the current page. Consider the following HTML code:

```html
<html>
<head>
<title>Victim Page</title>
</head>
<body>
<p>You were sent here by:<script>document.write(document.referrer);</script></p>
</body>
</html>
```

If the document referrer string contains the JavaScript code, this code will be executed in the current context. To exploit this type of vulnerability an attacker must have an intermediate page from which the user links to the vulnerable page. The attacker will send the link of the page the user is hosting, to the victim.

```html
http://www.attacker.com/domxsspage.html?<script>the malicious code</script>
```

Using the JavaScript redirects or the user interaction, the attacker links to the vulnerable page, causing the execution of the malicious code in the context of the page.
1.7 SESSION HIJACKING

Session hijacking takes effect directly within the web application into which it was injected by the XSS exploit. Therefore, the attacker possesses the same control over the application as the attacked user. By creating HTTP requests to the exploited application, the attacker is able to execute actions on the application using the victim’s current authentication state.

From the application’s point of view, all actions by the attacker executed through a session hijacking attack are indistinguishable from legitimate actions of the attack’s victim (i.e., by the authenticated user which accesses the exploited application). Thus, a session hijacking attack empowers the attacker to temporarily overtake the victim’s identity in respect of the exploited application. Session hijacking attacks may either require real-time interaction by the attacker, or be fully pre-scripted for automatic execution. The latter case is, for instance, used by XSS worms (Livshits and Lam 2008). All currently known XSS session hijacking attack methods can be assigned to one of the following different classes “Session fixation”, “browser hijacking” and “background XSS propagation”.

1.7.2 Session ID Fixation Attack

Web applications commonly employ a Session Identifier (SID) to track the authenticated state of a user. Every request that contains this SID is regarded as belonging to the authenticated user. By reading the SID value via JavaScript and communicating it to an attacker-controlled location by creating a cross-domain HTTP request, the attacker is able to obtain this authentication credential (De Ryck et al 2012). In a session ID fixation attack, the
vulnerable java script used for communicating the session ID allows one person to fixate another person's session identifier. Most session ID fixation attacks are web based, and most rely on session identifiers being accepted from URLs or header data. From this point, the stolen SID can be used to create further requests. As long as the SID is valid, the attacker can impersonate as the legitimate user (Klein 2005).

For example, consider the following link, where the attacker can fix the session ID to the user.

If a user clicks this specially crafted link in an email or any other means, an attacker fixed session id will be set to the user. When a user clicks on this link, it will redirect him to the authentication page. When a user gives his own authentication credential and logs in his page, the attacker session id will be fixed to a user. Both the attacker and user will be in the same session; so, the session fixation attack will occur.

1.7.3 Browser Hijacking

This method of session hijacking does not require the communication of the SID over the internet. The whole attack takes place in the victim’s browser. Modern web browsers provide the XMLHttpRequest object, which can be used to place the GET and POST requests to URLs that satisfy the same origin policy. Instead of transferring the SID or other authentication credentials to the attacker, the “browser hijacking” attack uses this ability to place a series of HTTP requests to the web application (Johns
The application’s server cannot differentiate between the regular, user initiated requests and those placed by the script.

Thus, the malicious script is capable of acting under the identity of the user, and commits arbitrary actions on the web application. In 2005, the so-called “Samy is my hero” worm employed this technique to create a self replicating JavaScript worm that infected approximately one million profiles on the website myspace.com (Jim et al 2007b). This attack does not depend on the availability of the XMLHttpRequest object. It can also be executed using hidden iframes (Lamarre 2005).

1.7.4 Background XSS Propagation

Usually not all pages of a web application are vulnerable to the XSS. For creating an XSS attack, it is sufficient that the user visits only one vulnerable page in which a malicious script has been inserted. However, other attack scenarios require the existence of a JavaScript on a certain webpage to work. For example, even when credit card information has been submitted it is seldom displayed in the web browser. In order to steal this information a malicious script would have to access the HTML form that is used to enter it. Consider the following scenario: Webpage A of the application is vulnerable against XSS whereas webpage B is not. Furthermore, webpage B is the page containing the credit card entry form. To steal the credit card information, the attacker would have to propagate the XSS attack from page A to page B (Johns 2006) as shown in Figure 1.7. This background XSS propagation can be possible by attackers through iframe and popup windows.
1.7.4.1 Propagation via iframe inclusion

In this case, the XSS replaces the displayed page with an iframe that takes over the whole browser window. Furthermore, the attacking script causes the iframe to display the attacked webpage, thus creating the impression that nothing has happened. From now on every user navigation is done inside the iframe. While the user keeps on using the application, the attacking script is still active in the document that contains the iframe. As long as the user does not leave the application’s domain, the malicious script is able to monitor the user’s surfing and to include further scripts in the web pages that are displayed inside the iframe (Johns 2006).

1.7.4.2 Propagation via pop under windows

A second way of XSS propagation can be implemented using “pop under” windows. The term “pop under” window denotes the method of
opening a second browser window that immediately sends itself to the background. On sufficiently fast computers, users often fail to notice the opening of such an unwanted window. The attacking script opens such a window and inserts the script code in the new window’s body. The new window has a link to the DOM tree of the original document (the father window) via the window.opener property. This link stays valid as long as the domain property of the father window does not change, even after the user resumes navigating through the web application. The script that was included in the new window is therefore able to monitor the user’s behaviour and include arbitrary scripts in the web pages of the application that are visited during the user’s session (Johns 2006).

1.8 WEB SERVICES FOR WEB APPLICATION SECURITY

Web Services are self-contained, reliable, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains (Guan et al 2005). These applications can be local, distributed, or Web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML (Dezhgosha et al 2005). So, web services allow a user to expose the functionality of the existing code over the network. Once it is exposed on the network, other applications can use the functionality. It allows different applications to talk to each other, and share data and services among themselves (Liu et al 2010). In addition, web services use standardized industry standard protocol for the communication.

The adding of a new functionality to the existing web applications would be a time consuming process. In particular, additional application layer security methods are inevitable in online transactions based web applications. While at the running state, the integration of new additional security methods cannot be possible inside the application or business logic. The perfect
solution to integrate additional functionality with the existing web applications is web services, as shown in Figure 1.8.

![Figure 1.8 Web service for web application Security](image)

Since, web services are independent of the web application architecture and platform, they can be integrated with any type of web applications. Consequently, this Public Key Infrastructure (PKI) addresses only half the problem with regard to e-commerce applications, and some complementary technologies are required to address the authorization problem as well. Westphall (1999) proposed an authorization scheme for large-scale networks that involves programming models and tools represented by the Web, Java and Common Object Request Broker Architecture (CORBA). The authorization scheme is based on structures and concepts introduced in the Web, Java and CORBA for security. This proposed scheme was developed in order to simplify authorization policy implementation in these systems. These policies are based on well-known literature security models such as the access matrix model. These security models are adapted in
order to elaborate a feasible and practical authorization scheme, in which it is easy to define policies to be followed in large-scale distributed systems.

Hence, web services are perfect solutions to integrate an additional security layer in the web applications.

1.9 ORGANIZATION OF THE THESIS

Chapter 2 discusses the general approaches to detect the vulnerabilities in web applications and the existing solutions for code injection vulnerabilities in web applications and its limitations.

Chapter 3 describes the proposed and designed architecture of the “Design of Web Applications Secure System from Code Injection Vulnerabilities through Web Services (WAPS-CIVS)” and the modules of SQL Injection preventer System, XPath Injection preventer System, Cross-site Scripting attack prevention system and Session Hijacking preventer System.

Chapter 4 gives a detailed implementation of the SQL Injection preventer system and its modules. Further, it explains the evaluation details of the designed SQL Injection prevention system with the existing approaches.

Chapter 5 explains the implementation details of the XPath injection prevention system. In addition, the results of the designed XPath Injection prevention are compared with those of the existing prevention approaches.

Chapter 6 describes the Cross site scripting prevention engine architecture and the implementation details. The results of the prevention engine are compared with the existing solutions.
Chapter 7 explains the three different types of Session hijacking prevention systems, such as the session id fixation, browser hijacking preventer and background XSS propagation. The results of the Session hijacking prevention system are analysed.