**Unit - 3**

**How to find the vectors and why?**

Every GET and POST parameter is a potential target for input validation attacks. Altering argument values, whether they are populated from FORM data or generated by the application, is a trivial feat. The easiest points of attack are input fields in forms. Common fields are Login Name, Password, Address, Phone Number, Credit Card Number, and Search. Other fields that use drop-down menus should not be overlooked, either.The first step is to enumerate these fields and their approximate input type.

Don’t be misled that input validation attacks can only be performed against fields that the user must complete. Every variable in the GET or POST request can be attacked. The attack targets can be identified by performing an in-depth crawl of the application that simultaneously catalogs files, parameters, and form fields. This is often done using automated tools.Cookie values are another target. Cookies contain values that might never be intended for manipulation by a user, but they can still be injected into to perform SQL injection or other injection attacks.

The cookie is simply a specific instance of an HTTP header. In fact, any HTTP header is a vector for input validation attacks.Another example of HTTP header-targeted attacks includes HTTP response splitting, in which a legitimate response is prematurely truncated in order to inject a forged set of headers (usually cookies or cache-control, which do the maximum damage client-side).

Let’s take a closer look at HTTP response splitting. This attack targets applications that use parameters to indicate redirects. A good input validation routine would ensure that the value for the page parameter consists of a valid URL. Yet if arbitrary characters can be included, then the parameter might be rewritten

The original value of page has been replaced with a series of characters that mimics the HTTP response headers from a web server and includes a simple HTML string for “Hello, world!” The malicious payload is more easily understood by replacing the encoded characters: Content-Type: text/html

HTTP/1.1 200 OK

Content-Type: text/html

<html>Hello, world!</html>

The end result is that the web browser displays this faked HTML content rather than the HTML content intended for the redirect. The example appears innocuous, but a malicious attack could include JavaScript or content that appears to be a request for the user’s password, Social Security number, credit card information, or other sensitive information.

**Advanced Directory Traversal**

A normal HTTP request returns the HTML content of login.htm:

<HTML>

<HEAD>

<TITLE>GroupWise WebAccess Login</TITLE>

</HEAD>

<!login.htm>

..remainder of page truncated…

The first alarm that goes off is that the webacc servlet takes an HTML file (login.htt) as a parameter because it implies that the application loads and presents the file supplied to the User.html parameter. If the User.html parameter receives a value for a file that does not exist, then we would expect some type of error to occur. Hopefully, the error gives us some useful information.

**Navigating Without Directory Listings**

Canonicalization attacks allow directory traversal inside and outside of the web document root. Unfortunately, they rarely provide the ability to generate directory listings—and it’s rather difficult to explore the terrain without a map! However, there are some tricks that ease the difficulty of enumerating files.The first trick is to find out where the actual directory root begins. This is a drive letter on Windows systems and most often the root (“/”) directory on Unix systems. IIS makes this a little easier, since the top-most directory is “InetPub” by default.

**Common input injection attack :**

Let's examine some common input validation attack payloads. Even though many of the attacks merely dump garbage characters into the application, other payloads contain specially crafted strings.

* **Buffer Overflow**

Buffer overflows are less likely to appear in applications written in interpreted or high-level programming languages.

* **Canonicalization (dot-dot-slash)**

These attacks target pages that use template files or otherwise reference alternate files on the web server. The basic form of this attack is to move outside of the web document root in order to access system files, i.e., “../../../../../../../../../boot.ini”.

* **HTML Injection**

Script attacks include any method of submitting HTML-formatted strings to an application that subsequently renders those tags. The simplest script attacks involve entering <script> tags into a form field

* **Cross-site Scripting (XSS)**

Cross-site scripting attacks place malicious code, usually JavaScript, in locations where other users see it. Target fields in forms can be addresses, bulletin board comments, and so forth. The malicious code usually steals cookies, which would allow the attacker to impersonate the victim or perform a social engineering attack, tricking the victim into divulging his or her password.

* **Embedded Scripts**

Embedded script attacks lack the popularity of cross-site scripting, but they are not necessarily rarer. An XSS attack targets other users of the application. An embedded script attack targets the application itself.

* **Cookies and Predefined Headers**

Web application testers always review cookie contents. Cookies, after all, can be manipulated to impersonate other users or to escalate privileges. The application must read the cookie; therefore, cookies are an equally valid test bed for script attacks.

**Canonicalization :**

These attacks target pages that use template files or otherwise reference alternate files on the web server. The basic form of this attack is to move outside of the web document root in order to access system files, i.e., “../../../../../../../../../boot.ini”. The actual server, IIS and Apache, for example, is hopefully smart enough to stop thisIIS fell victim to such problems due to logical missteps in decoding URL characters and performing directory traversal security checks. Two well-known examples are the IIS Superfluous Decode (..%255c..) and IIS Unicode Directory Traversal (..%c0%af..) vulnerabilities.A web application’s security is always reduced to the lowest common denominator. Even a robust web server falls due to an insecurely written application. The biggest victims of canonicalization attacks are applications that use templates or parse files from the server. If the application does not limit the types of files that it is supposed to view, then files outside of the web document root are fair game. These technique succeeds against web servers when the web application does not verify the location and content of the file requested.

**Countermeasures :**

The best defense against canonicalization attacks is to remove all dots (.) from GET and POST parameters. The parsing engine should also catch dots represented in Unicode and hexadecimal.

Force all reads to happen from a specific directory. Apply regular expression filters that remove all path information preceding the expected filename. For example, reduce/path1/path2/./path3/file to /file.

Secure filesystem permissions also mitigate this attack. First, run the webserver as a least-privilege user: either as the “nobody” account on Unix systems or create a service account on Windows systems with the least privileges required to run the application.Limit the web server account so it can only read files from directories specifically related to the web application.

Move sensitive files such as include files (\*.inc) out of the web document root to a directory with proper access control.Ensure that anonymous Internet users cannot directly access directories containing sensitive files and that only users with proper authorization will be granted permission. This mitigates directory traversal attacks that are limited to viewing files within the document root. The server and privileged users are still able to access the files, but the user cannot read them.

**HTML Injection attack :**

Script attacks include any method of submitting HTML-formatted strings to an application that subsequently renders those tags. The simplest script attacks involve entering <script> tags into a form field. If the user-submitted contents of that field are redisplayed, then the browser interprets the contents as a JavaScript directive rather than displaying the literal value <script>. The real targets of this attack are other users of the application who view the malicious content and fall prey to social engineering attacks.

There are two prerequisites for this attack. First, the application must accept user input. This sounds obvious; however, the input does not have to come from form fields. We will list some methods that can be tested on the URL, but headers and cookies are valid targets as well. Second, the application must redisplay the user input. The attack occurs when an application renders the data, which become HTML tags that the web browser interprets.

* Cross-site Scripting (XSS)
* Embedded Scripts
* Cookies and Predefined Headers

**Cookies and Predefined Headers :**

Web application testers always review cookie contents. Cookies, after all, can be manipulated to impersonate other users or to escalate privileges. The application must read the cookie; therefore, cookies are an equally valid test bed for script attacks. In fact, many applications interpret additional information that is particular to your browser. The HTTP 1.1 specification defines a User-Agent header that identifies the web browser. You usually see some form of “Mozilla” in this string.

Applications use the User-Agent string to accommodate browser quirks (since no one likes to follow standards).

The application can’t determine our custom User-Agent string. If we view the source, then we see why this happens:

<BLOCKQUOTE>

<PRE>

<script>

</PRE>

</BLOCKQUOTE>

So, our <script> tag was accepted after all. This is a prime example of a vulnerable application. The point here is that input validation affects any input that the application receives.

**SQL Injection**

A SQL injection attack exploits vulnerabilities in input validation to run arbitrary commands in the database. It can occur when your application uses input to construct dynamic SQL statements to access the database. It can also occur if your code uses stored procedures that are passed strings that contain unfiltered user input. Using the SQL injection attack, the attacker can execute arbitrary commands in the database. The issue is magnified if the application uses an over-privileged account to connect to the database. In this instance it is possible to use the database server to run operating system commands and potentially compromise other servers, in addition to being able to retrieve, manipulate, and destroy data.

**Example of SQL Injection**

Your application may be susceptible to SQL injection attacks when you incorporate unvalidated user input into database queries. Particularly susceptible is code that constructs dynamic SQL statements with unfiltered user input. Consider the following code:

SqlDataAdapter myCommand = new SqlDataAdapter(

"SELECT \* FROM Users

WHERE UserName ='" + txtuid.Text + "'", conn);

Attackers can inject SQL by terminating the intended SQL statement with the single quote character followed by a semicolon character to begin a new command, and then executing the command of their choice. Consider the following character string entered into the txtuid field.

'; DROP TABLE Customers–

This results in the following statement being submitted to the database for execution.

SELECT \* FROM Users WHERE UserName=''; DROP TABLE Customers --'

This deletes the Customers table, assuming that the application's login has sufficient permissions in the database (another reason to use a least privileged login in the database). The double dash (--) denotes a SQL comment and is used to comment out any other characters added by the programmer, such as the trailing quote.

Note: The semicolon is not actually required. SQL Server will execute two commands separated by spaces.

Other more subtle tricks can be performed. Supplying this input to the txtuid field:

' OR 1=1–

builds this command:

SELECT \* FROM Users WHERE UserName='' OR 1=1–

Because 1=1 is always true, the attacker retrieves every row of data from the Users table.

**Countermeasures** **:**

* Perform thorough input validation. Your application should validate its input prior to sending a request to the database.
* Use parameterized stored procedures for database access to ensure that input strings are not treated as executable statements. If you cannot use stored procedures, use SQL parameters when you build SQL commands.
* Use least privileged accounts to connect to the database.

**Sub queries in SQL injection :**

Subqueries can retrieve information ranging from Boolean indicators (whether a record exists or is equal to some value) to arbitrary data (a complete record). Subqueries are also a good technique for semantic-based vulnerability identification. A properly designed subquery enables the attacker to infer whether a request succeeded or not.The simplest subqueries use the logical AND operator to force a query to be false or to keep it true:

AND 1=1

AND 1=0

Now, the important thing is that the subquery be injected such that the query’s original syntax suffers no disruption. Injecting into a simple query is easy:

SELECT price FROM Products WHERE

More complex queries that have several levels of parentheses and clauses with JOINs might not be as easy to inject with that basic method.

**UNION in SQL Injection :**

The SQL UNION operator combines the result sets of two different SELECT statements. This enables a developer to use a single query to retrieve data from separate tables as one record. The following is a simple example of a UNION operator that will return a record with three columns:

SELECT c1,c2,c3 FROM table1 WHERE foo=bar UNION

SELECT d1,d2,d3 FROM table2 WHERE this=that

A major restriction to the UNION operator is that the number of columns in eachrecord set must match. This isn’t a terribly difficult thing to overcome; it just requires some patience and brute-force.Column undercounts, where the second SELECT statement has too few columns, are easy to address. Any SELECT statement will accept repeat column names or a value.

**XPATH Injections :**

In addition to storing data in an RDBMS, web applications also commonly store data in an XML format. XPATH is the query language used to parse and extract specific data out of XML documents, and by injecting malicious input into an XPATH query, we can alter the logic of the query. This attack is known as XPATH injection.

Unlike SQL injection, there is no way to comment out parts of the query when using XPATH. Therefore, an attacker must inject additional logic into the query, causing it to return true when it otherwise may have returned false or causing it to return additional data.

A **dangerous example** of how an XPATH injection could be used to bypass authentication is based on the following code:String(//users/admins/[user/text()=' " + txtUser.Text + " '

and pass/text()=' "+ txtPass.Text +" '])

If the input is admin' or 1=1 or 'a'='b', the query will be:

String(//users/admins/[user/text()='admin' or 1=1 or 'a'='b'

and pass/text()=''])

The expression

user='admin' or 1=1 or 'a'='b' and pass/text()=' '

can be represented as

(A OR B) OR (C AND D)The logical operator AND has higher priority than OR, so if either A or B is true, the expression will evaluate to true irrespective of what (C AND D) returns. If the user input for the query, B is 1=1, which is always true, it makes the result of (A OR B) true. Thus the query returns true and the attacker is able to log in—bypassing the authentication mechanism with XPATH injection.

**Countermeasures :**

Like SQL injection, XPATH injection can be prevented by employing proper input validation and parameterized queries. No matter what the application, environment, or language, you should follow these best practices:

• Treat all input as untrusted, especially user input, but even input from your database or the supporting infrastructure.

Validate not only the type of data but also its format, length, range, and type (for example, a simple regular expression such as (/^"\*^';&<>()/) would find suspect special characters).

• Validate data both on the client and the server because client validation is extremely easy to circumvent.

• Test your applications for known threats before you release them.

This technique provides solid protection from XPATH injection, although it is not built in to the XPATH specification. The user input is not directly used while forming the query; rather, the query evaluates the value of the element in the XML document, and if it does not match the parameterized value, it fails gracefully. It is possible to extract an entire XML document through a web application that is vulnerable to XPATH injection attacks.With the increased adoption of techniques such as Ajax, RIA platforms such as FLEX, or Silverlight, as well as the adoption of XML services from organizations such as Google that rely heavily on the use of XML for everything from communication with backend services to persistence, now more than ever, we need to remain vigilant about the threats and risks created by these approaches.

**LDAP Injection :**

Another data store that should only accept validated input from an application is an organization’s X.500 directory service, which is commonly queried using the Lightweight Directory Access Protocol (LDAP). An organization allowing unvalidated input in the construction of an LDAP query is exposed to an attack known as LDAP injection. The threat posed allows an attacker to extract important corporate data, such as user account information, from the LDAP tree.By manipulating the filters used to query directory services, an LDAP injection attack can wreak havoc on single sign-on environments that are based on LDAP directories.

LDAP directory services are critical repositories for managing an organization’s user data. If a compromise were to occur, personally identifiable information will almost certainly be exposed and may allow for successful authentication bypass attacks. Be sure to review all user input that interacts with LDAP directory services.