Attacking Data Stores

- Injecting into Interpreted Contexts
  → Most Data Store languages are are interpreted. And in therie execution, they use the data supplied by the user to be
    substituted in a query or command and actions being taken on said query's results.
  → By injecting a special payload the attacker can break out of the data context and inject payloads that will be processed
    as code by the data store. Hence compromising the application.

  → Bypasssing a Login
    ⇒ The process by which data stores are accessed are findamentally same for priv and unpriv users.
    ⇒ The app functions as a discretionary access control to the data store.
    ⇒ If security-sensitive application logic is controlled by the results of a query, an attacker ca potentially modify the query
      to alter logic of the app.
    ⇒ Ex:
      • SQL statement for login:
        ◇ SELECT * FROM users WHERE username = ‘marcus’ and password ‘secret’
        ◇ if attacker injects admin' --
        ◇ SELECT * FROM users WHERE username='admin' --' AND password = ‘fpp', notice the -- sequence renders the later half of the statement useles and the attacker will be able to login as the admin
        ◇ Classic payload = ' OR 1=1 --
        ◇ Basically renders any statement to be true

- Injecting into SQL
  → Almost every web application employs a databaseto store various kinds of information.
  → This means accessing information within DB is dne by SQL (Structured Query Language)
  → Since SQL queries are constructed using user input, doing this in an unsafe manner can result in a vuln
  → In the worst case it can allow the attacker to take complete control of the server

  → Exploiting a Basic Vuln
    ⇒ Start by checking for ' character.
    ⇒ If Verbose error message is seen,then this application is wide open for SQL Injection.
    ⇒ Use either ‘ OR 1=1 -- or balacing quotes to show proof of concept.

  → Injecting into Different Statements
    ⇒ SELECT Statements - Common for SQL injection vulns
      • Often used in searching, login etc. Entry point is the WHERE clause.
      • Sometimes affects other parts such as the ORDER BY clause.

    ⇒ INSERT Statemenets - Used in Creation of rows
      • Commonly used in creatin of data

• Attacker can inject payloads to create data in the DB. However it has be in proper syntax to satisfy the VALUES clause
• Blind Injections can be leveraged by retrieving string data by savinf them into a parameter that will be displayed to the user. Hence the injection's results can be reviewed on the user's profile
• To fgure out the no. of columns, we start with one payload and keep incrementing until the desired row is crated

⇒ UPDATE Statements
- Used in updation of rows
- Crafted input can be used to update values and hence lead to takeover of the app

⇒ Delete Statements
- Similar caution applies as with UPDATE statement

→ **Finding SQL Injection Bugs**
⇒ Detection is difficult and often may be subtle.
⇒ Steps to verify
- Injecting into String Data
  ◇ Exploiting any SQL injection vuln, you need to break out of the SQL query's quotation marks
  ◇ Inject ' and observe behaviour. If any,balance and continue exploitation.
  ◇ Proceed to exploit
- Injecting into Numeric Data
  ◇ No ' is used for numeric Data.
  ◇ Try injecting a mathematical expression equal the original value.
  ◇ Encode the HTTP characters in URL otherwise they'll tend to malform your payloads.
- Injecting into the Query Strcuture
  ◇ Injecting params into the query structure can not be detected by normal payloads or automated fuzzing.
  ◇ Hence testing these maually is the only way you can detect and exploit the vuln

→ Fingerprinting the Database
⇒ Using Concat statements to know the database used by the app.
- Oracle: 'serv' || 'ices'
- MS-SQL: 'serv' + 'ices'
- MySQL: 'serv' 'ices' (note the space)

⇒ If you are injecting into numeric data,
- Oracle: BITAND(1,1)-BITAND(1,1)
- MS-SQL: @@PACK_RECIEVED-@@PACK_RECIEVED
- MySQL: CONNECTION_ID()- CONNECTION_ID()

⇒ MySQL : Comments /* !32302 and 1=0*/

→ The Union Operator
⇒ Used to combine results of SELECT stmts.
⇒ Restraints
- The two sets must have same structure. i.e Same datatype, same no. of columns etc.

- Attacker should know names of tables and its columns

⇒ Guesswork
- Compatible dtypes : Use NULL i.e SELECT NULL (for oracle, USE SELECT NULL FROM DUAL)
- Traps error messages: Extrpolate from returned results
- Even a single string param is enough to compromise the DB info

→ Extracting Data with UNION

⇒ Use information_schema.columns to obtain useful information

→ Bypassing Filters

⇒ Avoiding Blocked Characters
- Use inbuilt string functions to bypass filters
  - ◇ Ex: SELECT ename, sal FROM emp where ename=CHR(109)||CHR(97)||CHR(114)||CHR(99)||CHR(117)||CHR(115)
- If Comment is blocked, use String balancing
  - ◇ Ex: Instead of ' OR 1=1 -- , use ' or 'a'='a

⇒ Circumventing Simple Validation
- Blacklists can be bypassed
- Try and use canonical lists to bypass these

⇒ Using SQL comments
- Inline comments can be interpreted as whitepaces.and hence can be used to bypass blacklists.
  - ◇ Ex: SEL/*foo*/ECT username,password FR/*foo*/OM users

→ Second Order SQL Injection

⇒ A particularly weird application is second order SQLi

⇒ Wherein initially the data is stored safely but when retrieved, it can cause problems.

⇒ One such example is the doubling up strategy used in authentication.

→ Advanced Exploitation

⇒ Retrieving Data as Numbers
- Often string inputs are handled correctly by the application. In these cases numeric data can be used
- Two key functions
  - ◇ ASCII
  - ◇ SUBSTRING or SUBSTR
- Inducing time delay
- Going Out-of-band

→ **Beyond SQLi : Escalating the Attack**

⇒ Shared database as a vector to other apps

⇒ OS compromise

⇒ Gaining Network Access

⇒ Network Connection back to your infra to transfer and ump lot of data

⇒ Extend DB functions

→ **Preventing SQLi**

⇒ Partially Effective Measures

- Doubling Up
  - ◇ Fails in Numeric data and and cause Second Order SQLi
- Stored Procedure to access data
  - ◇ Fails just as miserably as an unprotected system and inundates an overhead on dev time.
- Parameterized Queries
  - ◇ Done in two steps,
    - ▪ The application specifies query's structure, leaving placholders for input
    - ▪ The app specifies the contents of each placeholder
  - ◇ Some important provisos
    - ▪ Use in every query
    - ▪ Proper paramertization
    - ▪ Cannot be used to specify table and column names
    - ▪ Cannot be used in clauses
- Defense in Depth
  - ◇ Use lowest possible priv
  - ◇ Remove unnecessary functions that can be leveraged
  - ◇ Timely Patching

- Injecting into NoSQL

→ Any datastore that defers from the standard relational model

→ Used key value pairs to store data

→ Allows presence of hierarchy instead of a flat database schema

→ Common query methods in NoSQL,

⇒ Key/ Value lookup

⇒ XPATH

⇒ Languages such as JS

→ Injecting into MongoDB

⇒ Use the languages's innate features to exploit.

⇒ Such as // or \ to bypass functions

→ **Injecting into XPATH (XML Path Language)**

⇒ XPATH is an interpreted language used to retrieve data from xml docs.

⇒ Subverting App Logic

- Similar payloads such as ' or 'a'='a can be used in fetching statements

⇒ Informed XPath Injection

- Similar to SQLi using substring and ASCII functions can lead an attacker to gain access to data.

⇒ Blind XPath Injection

- Using meta functions and cycling through data can lead an attacker to gain access to data

character by character.

→ **Finding XPath Injection Flaws**

⇒ Feeding SQLi payloads can result in finding XPATH Injections =.

• Ex: Using ` and ’ -- can invalidate syntax and show errors

⇒ Similar to SQLi payload try using ` or ’1'=1' and its variants

⇒ Hence situation where SQLi payloads cause error but exploitation is not possible shoudl be trated tentatively as XPATH injection vulns, but to confirm you must investigate and prove it.

→ Prevention of XPath Injection

⇒ Input sanitzation should be implemented alongside a whitelist to substitute values in a query.

- Injecting into LDAP

→ The LDAP (Lightweight Directory Access Protocol) is used to access directory services over a network.

→ Not readily exploitable

⇒ Seacrh filter can't be easily exploited

⇒ Data is passed to API's as a paarmeters

⇒ Vulns have to be exploited blind

→ Exploiting LDAP injection

⇒ Disjunctive Queries

• Using wildcards to match all data

⇒ Conjective Queries

• Using batch queries

• Using null bytes to `comment out`

→ **Finding LDAP injections**

⇒ Cannot be error based as errors lead to 500 HTTP status code

• Nevertheless

◇ Try entering * as search term; if large no of results are shown you might be looking at a LDAP query

◇ Try entering)))))))) this closes out the search and main brackets hence if the unvaliated brackets produce an error, indicates LDAP injection.

◇ Try using the cn attribute to probe the app

→ **Preventing LDAP Injection**

⇒ If neccessary, perform insertion only in simple terms subjected to strict input validation

⇒ Should be checked against a a white list

⇒ Block chars such as ( ) ; , * } & = and Null byte.

⇒ Any input that doesnt match should be rejected and not sanitized.