I'm not a robot

Sql injection cheat sheet owasp

Given text here Application injection flaws are a common issue, affecting many applications due to their accessibility and vulnerability. With most apps sourced from external sources, closed-source ones require a distinct approach to prevent injection flaws. These occur when untrusted data is sent to an interpreter, often in SQL queries, LDAP queries, or OS commands. While easier to detect in code, attackers use scanners and fuzzers to find them during testing. The best solution lies in fixing the issue in source code or redesigning parts of applications. However, if source code is unavailable or unfeasible, virtual patching may be necessary. Applications can be categorized into three types: New Application, Productive Open Source Application, and Productive Closed Source Application. Each type requires different actions to prevent injection flaws. A new application's design phase is an ideal time for prevention, while open-source applications can be adapted easily. Closed-source applications, however, pose a challenge due to their inflexibility. Injection attacks target various technologies, including SQL queries, LDAP queries, XPath queries, and OS commands. Query languages like SQL Injection allow attackers to modify database queries, compromising sensitive data. Other forms of injection include LDAP, SOAP, XPath, and REST-based queries, which can be vulnerable to attack. SQL Injection is a type of injection attack where an attacker injects SQL code into data-plane input to execute predefined SQL commands. This attack can read sensitive data, modify database data, or even shut down the DBMS. It can also recover file content or write files to the system file system. There are three classes of SQL Injection attacks: Inband, which extracts data using the same channel used for injection; OutofBand, where data is extracted from a separate channel; and Remote, where attackers execute commands on the server remotely. The attack types discussed in this article include: In-band: Data is presented directly in the web page after retrieval. Out-of-band: Data is retrieved through a different channel, such as an email with query results. Inferential or Blind: No actual data transfer occurs, but the tester reconstructs information by sending specific requests and observing DB Server behavior. Testing for issues involves: Code review: Checking queries to ensure they use prepared statements and sanitized data. Auditing stored procedures for uses of sp_execute, execute, or exec. Automated exploitation using tools like SQLMap. Key concepts include: Stored Procedure Injection: User input must be sanitized to prevent code injection. Time delay Exploitation technique: Measuring time delays to infer query results in Blind SQL Injection situations. Out-of-band Exploitation technique: Using DBMS functions to deliver injected query results out of band. Defense options include: Prepared Statements (with Parameterized Queries): Preventing attackers from changing query intent. Other defense options are not explicitly mentioned. The distinction between prepared statements and stored procedures lies in how SQL code is handled. Prepared statements have the code defined outside the database, whereas stored procedures store their code within the database itself, called from applications. Both techniques can effectively prevent SQL injection attacks, making them equally suitable for most organizations' needs. However, while stored procedures are not inherently safe, certain constructs implemented correctly have a similar effect to parameterized queries in preventing SQL injection. This is particularly true when it comes to standard stored procedure programming practices. Note that the effectiveness of stored procedures against SQL injection heavily relies on their implementation, specifically ensuring no dynamic unsafe SQL generation occurs. In contrast, Defense Option 3 involves allowing-list input validation for parts of SQL queries where bind variables aren't appropriate, such as table or column names and sort order indicators. Ideally, these values should come from code rather than user parameters. However, when using user parameter values, it's crucial to map them to legal/expected table or column names to prevent unvalidated user input from ending up in the query. If time allows, this could indicate a symptom of poor design and necessitate a full rewrite. Defense Option 4 is only recommended as a last resort due to its frailty compared to other defenses. This approach involves escaping all user-supplied input before placing it in a query. It's usually advised for legacy code when implementing input validation isn't cost-effective. The example provided uses Java's PreparedStatement and CallableStatement interfaces, showcasing safe ways to execute database queries using parameterized queries or stored procedures. These techniques are supported across various languages, including Cold Fusion and Classic ASP, making them effective measures against SQL injection attacks. An attacker exploits web applications that build LDAP statements based on user input, similar to SQL Injection. If an application doesn't sanitize user input properly, it's possible to modify LDAP statements, leading to unauthorized access and content changes within the LDAP tree. To learn more about LDAP Injection attacks, visit LDAP injection. These attacks are common due to two factors: the lack of safer parameterized LDAP query interfaces and the widespread use of LDAP for user authentication. Here's how to test for this issue: 1. During code review, check if LDAP queries escape special characters. 2. Use automated tools like OWASP ZAP with its Scanner module to detect LDAP injection issues. 3. Escape all variables using the correct LDAP encoding function. LDAP stores names based on DNs (distinguished names), which serve as unique identifiers, similar to usernames. These DNs have certain special characters: `\ # + < > , ; " =` and leading/trailing spaces. DNs point to single entries, like rows in a RDBMS, with attributes analogous to columns. To search for users with specific attributes, use search filters written in Polish notation (prefix notation). Example: `(&(ou=Physics)(| (manager=cn=Freeman Dyson,ou=Physics,dc=Caltech,dc=edu) (manager=cn=Albert Einstein,ou=Physics,dc=Princeton,dc=edu)))` When building LDAP queries in application code, you MUST escape any untrusted data added to LDAP queries. There are two forms of LDAP escaping: for search filters and for DNs. Example Java code shows safe LDAP escaping: ```java public String escapeDN(String name) { final char[] META_CHARS = {'+', '"', '<', ';', '/'}; String escapedStr = new String(name); // Backslash is both a Java and an LDAP escape character, so escape it first escapedStr = escapedStr.replaceAll("\\\\\\\\","\\\\\\\\"); // Positional characters - see RFC 2253 escapedStr = escapedStr.replaceAll("\^#","\\\\\\\\#"); escapedStr = escapedStr.replaceAll("\^ | $","\\\\\\\\ "); for (int i=0 ; i < META_CHARS.length ; i++) { escapedStr = escapedStr.replaceAll("\\\\" + META_CHARS[i],"\\\\\\\\" + META_CHARS[i]); } return escapedStr; } ``` A vulnerability in web applications arises when user-input code is executed without proper validation and sanitization. This can lead to XPath injection, a technique that allows an attacker to subvert application logic and gain local access. Scripting languages used in web apps often have eval calls that execute code at runtime, making them susceptible to code injection attacks if unvalidated user input is used. Operating System Commands ---------- OS command injection occurs when users supply OS commands through a web interface, allowing the execution of commands on the server. This vulnerability can be exploited by appending operating system commands to URL query parameters, as demonstrated in the example: ` . To prevent OS command injection: 1. **Parameterization**: Use structured mechanisms that enforce data-command separation and provide quoting and encoding. 2. **Input Validation**: Validate both command values and their arguments. Validation Degrees ---------- * Command validation: Compare against a list of allowed commands. * Argument validation: * Positive or allowlist input validation: Define explicitly allowed arguments. * Allow-list Regular Expression: Specify good characters using regular expressions. Given article text here ProcessBuilder b = new ProcessBuilder("C:\DoStuff.exe -arg1 -arg2"); ProcessBuilder pb = new ProcessBuilder("TrustedCmd", "TrustedArg1", "TrustedArg2"); Map env = pb.environment(); pb.directory(new File("TrustedDir")); In web applications, network daemons like SMTP, IMAP, FTP can be vulnerable to command injection. Injection Prevention Rules are essential. Rule #1 is to perform proper input validation. Rule #2 recommends using a safe API. If a parameterized API is not available, contextually escape special characters. SQL injections allow attackers to read sensitive data or modify database operations. Threat modeling shows that SQL injection attacks can spoof identity, tamper with data, void transactions, or become administrators. J2EE and ASP.NET applications are less susceptible to SQL injection attacks due to their inherent security features. However, it is essential to consider the attacker's skills and imagination, as well as defense in depth countermeasures, to mitigate the severity of these attacks. SQL injection occurs when unintended data enters a program from an untrusted source, dynamically constructing a SQL query. The main consequences include loss of confidentiality, authentication bypass, authorization manipulation, and integrity breaches. Key risk factors include platform vulnerability (SQL or web), language, and the presence of user input fields that can be exploited to execute malicious SQL code. Examples of SQL injection attacks include injecting meta characters into data input to place SQL commands in the control plane, exploiting the lack of distinction between the control and data planes made by SQL. Best practices for preventing SQL injection vulnerabilities include using parameterized queries, validating user input, and implementing defense in depth countermeasures. The user's name was being used to filter database entries, but a malicious input could bypass this restriction. When the code concatenated user input with a query string, it became vulnerable to SQL injection attacks. An attacker could inject arbitrary SQL commands by entering special characters like single-quotes and semicolons in the itemname field. This allowed the attacker to execute unauthorized queries or even delete database entries. The issue arose because the code did not properly escape user input, leaving it open to exploitation. However, using parameterized SQL statements can offer more security guarantees with less maintenance. This approach involves treating user input as parameters rather than part of the query itself. By doing so, the risk of SQL injection attacks is significantly reduced, and database security can be enhanced. Given article text here manually escaping characters in input to SQL queries can help but it will not make your application secure from sql injection attacks another solution commonly proposed for dealing with sql injection attacks is to use stored procedures although stored procedures prevent some types of sql injection attacks they fail to protect against many others for example a plsql procedure is vulnerable to the same sql injection attack shown in the first example Given text here Microsoft SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE NULL END FROM dual PostgreSQL 1 = (SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/(SELECT 0) ELSE NULL END) MySQL SELECT IF(YOUR-CONDITION-HERE,(SELECT table_name FROM information_schema.tables),'a') You can potentially elicit error messages that leak sensitive data returned by your malicious query. Microsoft SELECT 'foo' WHERE 1 = (SELECT 'secret') > Conversion failed when converting the varchar value 'secret' to data type int. PostgreSQL SELECT CAST((SELECT password FROM users LIMIT 1) AS int) > invalid input syntax for integer: "secret" MySQL SELECT 'foo' WHERE 1=1 AND EXTRACTVALUE(1, CONCAT(0x5c, (SELECT 'secret'))) > XPATH syntax error: '\secret' Batched (or stacked) queries You can use batched queries to execute multiple queries in succession. Note that while the subsequent queries are executed, the results are not returned to the application. Hence this technique is primarily of use in relation to blind vulnerabilities where you can use a second query to trigger a DNS lookup, conditional error, or time delay. Oracle Does not support batched queries. Microsoft QUERY-1-HERE; QUERY-2-HERE QUERY-1-HERE; QUERY-2-HERE PostgreSQL QUERY-1-HERE; QUERY-2-HERE MySQL QUERY-1-HERE; QUERY-2-HERE With MySQL, batched queries typically cannot be used for SQL injection. However, this is occasionally possible if the target application uses certain PHP or Python APIs to communicate with a MySQL database. Time delays You can cause a time delay in the database when the query is processed. The following will cause an unconditional time delay of 10 seconds. Oracle dbms_pipe.receive_message(('a'),10) Microsoft WAITFOR DELAY '0:0:10' PostgreSQL SELECT pg_sleep(10) MySQL SELECT SLEEP(10) Conditional time delays You can test a single boolean condition and trigger a time delay if the condition is true. Oracle SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 'a'||dbms_pipe.receive_message(('a'),10) ELSE NULL END FROM dual Microsoft IF (YOUR-CONDITION-HERE) WAITFOR DELAY '0:0:10' PostgreSQL SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN pg_sleep(10) ELSE pg_sleep(0) END MySQL SELECT IF(YOUR-CONDITION-HERE,SLEEP(10),'a') DNS lookup You can cause the database to perform a DNS lookup to an external domain. To do this, you will need to use Burp Collaborator to generate a unique Burp Collaborator subdomain that you will use in your attack, and then poll the Collaborator server to confirm that a DNS lookup occurred. Oracle (XXE) vulnerability to trigger a DNS lookup. The vulnerability has been patched but there are many unpatched Oracle installations in existence: SELECT EXTRACTVALUE(xmltype('