


PURDUE UNIVERSITY

## Computer Security CS 426 Lecture 23

### SQL Injection Attacks

**Elisa Bertino**  
Purdue University  
IN, USA  
bertino@cs.purdue.edu



Center for Education and Research  
in Information Assurance and Security

PURDUE UNIVERSITY


## SQL Injection

- SQL injection attacks are ranked as the second most common form of attack on web applications for 2006 in CVE (Common Vulnerabilities and Exposures list - CVE <http://cve.mitre.org/>)
- CardSystems, credit card payment processing ruined by SQL Injection attack in June 2005
  - 263,000 credit card # stolen from its DB
  - credit card # stored unencrypted, 40 million exposed

PURDUE UNIVERSITY

## SQL Injection: description

- SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application
- A SQL injection attack involves the alteration of SQL statements issued by the application
- The vulnerability occurs when the application does not check the user input OR incorrectly filters user input for string literal escape characters embedded in SQL statements OR when user input is not strongly typed



```

graph LR
    form[form] --> Application[Application]
    Application -- SQL query --> DB[(DB)]
  
```

PURDUE UNIVERSITY

## SQL Injection

- Injection mechanisms:
  - Through the user input (e.g., HTTP GET/POST parameters)
  - Through cookies:
    - If a Web application uses the cookie's contents to build SQL queries, an attacker could easily submit an attack by embedding the "malicious" input in the cookie

## SQL Injection attack

Attack intent:

1. *Identifying application injectable parameters:* The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQL injection
2. *Performing database finger-printing:* The attacker wants to discover the type and version of database that a Web application is using
  - Different DBMS respond differently to different queries and attacks. This information can be used to "fingerprint" the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database specific attacks

## SQL Injection attack

Attack intent:

3. *Determining the database schema:* To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types
4. *Extracting data:* These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and valuable to the attacker

## SQL Injection attack

Attack intent:

5. *Adding or modifying data:* The goal of these attacks is to add or change information in a database
6. *Performing denial of service:* These attacks are performed to shut down the database, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category
7. *Evading detection:* This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms

## SQL Injection attack

Attack intent:

8. *Bypassing authentication:* The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. This way the attacker assumes the rights and privileges associated with another application user
9. *Executing remote commands:* These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.
10. *Performing privilege escalation:* These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

## SQL Injection attack

Attack phases:

1. Explore (probing)
2. Experiment
3. Try to exploit the vulnerability (ies) discovered in 1 and 2 above (attack)

## SQL Injection attack *Explore*

### 1. Explore (probing)

Determine the functionalities exposed by the application

- By sniffing and analyzing the traffic to/from a given web service
- By examining the landing HTML page of the web site

## SQL Injection attack *Experiment*

### 2. Experiment

- Goal: determine user-controllable input susceptible to injection
- How:
  - by attempting to inject characters that have special meaning in SQL (such as a single quote character, a double quote character, two hyphens, a parenthesis, etc.)
    - inject input through text fields or through HTTP GET parameters
    - modify HTTP POST parameters, hidden fields by using a web application debugging tool
  - By analyzing the response/output of the application (which often returns back to the caller the DBMS error response unfiltered): a typical error resulting from an injection would look like: "You have an error in your SQL Syntax. Check your manual for the right syntax to use *near* ' FROM db\_users.user\_table

## SQL Injection attack *Experiment*

- Perform the attack through malicious SQL input injection
- SQL injections attacks can be further classified in:
  - First order SQL injection attack
    - the results of the injection are observed immediately in the form of an error message or a successful query resulting in attaining the attacker's intent
  - Second order SQL injection attack
    - the results of a successful injection are observed when a different query is executed or a different user accesses the application

### SQL Injection attack example 1

- Example 1: Incorrectly filtered escape characters (like single quote)  
**SQL statement := "SELECT \* FROM users WHERE name = " + userName + ""; "**
- If a malicious user puts in the username variable the following string:
  - a' or 't='t
- Then the following SQL statement is generated:
  - **SELECT \* FROM users WHERE name = 'a' OR 't='t';**
- If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of 't='t' is always true
- The problem arises because no control is made on the input string

### SQL Injection attack example 2

- Example 2: in this case the SQL engine allows the execution of multiple commands in the same SQL query  
**SQL statement := "SELECT \* FROM users WHERE name = " + userName + ""; "**
- If a malicious user puts in the *username* variable the following string:
  - a';DROP TABLE users; SELECT \* FROM data WHERE name LIKE '%'
- Then the following SQL statement is generated:
  - **SELECT \* FROM users WHERE name = 'a';DROP TABLE users; SELECT \* FROM DATA WHERE name LIKE '%';**

### SQL Injection attack example 3

- Example 3: a user supplied field is not strongly typed or is not checked for type constraints by the application  
**SQL statement := "SELECT \* FROM data WHERE id = " + a\_variable + "";"**
- With this statement the developer intended a variable to be a number correlating to the "id" field. However, if it is in fact a string, the end users may manipulate the statement as they choose, thereby bypassing the need for escape characters
- For example, if the malicious input is injected:
  - 1;DROP TABLE users
- Then the following SQL statement is generated:
  - **SELECT \* FROM DATA WHERE id = 1;DROP TABLE users;**

### Second order SQL injection attack example

- Even if an application always escapes single - quotes, an attacker can still inject SQL as long as data in the database is re-used by the application
- An attacker might register with an application, creating a username
  - Username: admin'--
  - Password: password
  - Suppose that the application correctly escapes the single quote, resulting in an 'insert' statement like this:
    - **insert into users values( 123, 'admin'--, 'password', 0xffff )**

Note: the application escapes the '

## Second order SQL injection attack example

- Suppose that the application allows users to change their password. The ASP script code first ensures that the user has the 'old' password correct before setting the new password. The code might look like this:

```
username = escape( Request.form("username") );
oldpassword = escape( Request.form("oldpassword") );
newpassword = escape( Request.form("newpassword") );
var rso = Server.CreateObject("ADODB.Recordset");
var sql = "select * from users where username = " + username + " and password = " +
oldpassword + """;
rso.open( sql, cn );
if (rso.EOF)
{
...
rso("username") is the username retrieved from the 'login' query
```

## Second order SQL injection attack example

- The query to set the new password might look like this:
  - sql = "update users set password = " + newpassword + " where username = " + rso("username") + ""
- Given the username admin'--, the query produces the following query:
  - update users set password = 'password' where username = 'admin'--'
- The attacker can set the admin password to the value of his choice, by registering as a user called admin'--.**

Note that the double hyphens are used in SQL to denote comments. Everything after the -- until the end of the line is considered a comment not part of the SQL

## SQL Injection countermeasures

- Requirement definition phase:
  - A non-SQL style database which is not subject to this flaw may be chosen
- Design phase:
  - Duplicate any filtering done on the client-side on the server side
  - Follow the principle of least privilege when creating user accounts to a SQL database. Users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data

## SQL Injection countermeasures

- Implementation phase:
  - Disallow meta-characters rather than escaping them
  - Narrowly define the set of safe characters based on the expected value of the parameter in the request
  - White-list style checking on any user input that may be used in a SQL command
  - Build the SQL query strings using parameterized statements that bind variables. Parameterized statements that do not bind variables can be vulnerable to attack

## SQL Injection countermeasures

- Input validation
  - Complex matter
  - Often overlooked in the implementation phase since it does not add functionalities
- Approaches to input validation:
  - Transform (“filtering”) the input
  - *Blacklisting*: reject input that is known to be “bad”, **BUT**
    - Known “bad input” may change over time as new attack techniques develop
  - *Whitelisting*: accept only input that is known to be “good”

## SQL Injection countermeasures filtering characters

- Input validation – filtering characters
  - Filter out characters like:
    - single quote, double quote,
    - slash, back slash,
    - semi colon,
    - extended character like NULL,
    - carriage return,
    - new line, etc,
  - in all strings from:
    - input from users
    - parameters from URL
    - values from cookie

## SQL Injection countermeasures reject “bad” input

- The application must act deterministically when it receives invalid characters from a user.
- For example, applications should patently reject users that submit two dash characters (--) or a semi-colon character (;) as part of their login name or password, and a high severity alert should be sent to application administrators.
- However, this somewhat harsh response may not be appropriate when an application receives a single apostrophe character as part of a person name (e.g., O'Brien) or street address that is submitted by an authenticated user who is entering package shipping information. Nonetheless, the application should behave as intended, and a notification should be sent to application administrators if the submitted apostrophe was not handled properly by the application.
- Invalid characters should also consider numeric parameters

## SQL Injection countermeasures reject “bad” input

**Example: disallow select/insert/update/delete/drop/--' in the user input**

```
function validate_string( input )
  known_bad = array( "select", "insert", "update", "delete",
    "drop", "--", "'" )
  validate_string = true
  for i = lbound( known_bad ) to ubound( known_bad )
    if ( instr( 1, input, known_bad(i), vbtextcompare ) <> 0 ) then
      validate_string = false
    exit function
  end if
next
end function
```

## SQL Injection countermeasures reject “bad” input

- Problem: when combining the filtering of data with validation of character sequences the attacker could still evade the detection
- Example:
  - A known bad input filter detects '--', 'select' and 'union' followed by a filter that removes single-quotes
  - The attacker could specify an input:
    - `uni'on sel'ect @@version--`
  - Since the single-quote is removed after the 'known bad' filter is applied, the attacker can simply intersperse single quotes in his known-bad strings to evade detection

## SQL Injection countermeasures accept only “good” input

- Accept only input that is know to be “good”
- Narrowly define the set of safe characters based on the expected value of the parameter in the request
  - Example:

```
good_password_chars =
"abcdefghijklmnopqrstuvwxyABCDEFGHJKLMNOPQR
STUVWXYZ0123456789"
```

## SQL Injection countermeasures accept only “good” input

```
function validatepassword( input )
good_password_chars =
"abcdefghijklmnopqrstuvwxyABCDEFGHJKLMNOPQR
STUVWXYZ0123456789"

validatepassword = true
for i = 1 to len( input )
c = mid( input, i, 1 )
if ( InStr( good_password_chars, c ) = 0 ) then
validatepassword = false
exit function
end if
next
end function
```

## SQL Injection countermeasures accept only “good” input

- Good character set can implicitly defined through *regular expressions*
- *RegExp* – pattern to match strings against
- Ex: suppose you have a *month* parameter which must be non-negative integer
  - RegExp: `^[0-9]*$` - 0 or more digits, safe subset
  - The `^`, `$` match beginning and end of string
  - `[0-9]` matches a digit, `*` specifies 0 or more

## SQL Injection countermeasures: parameterized statements

- Parameterized statements use parameters (sometimes called placeholders or bind variables) instead of directly embedding user input in the SQL statement
- In many cases, the SQL statement is fixed. The user input is then assigned (bound) to a parameter
- An example using Java and the JDBC API:
  - `PreparedStatement prep = conn.prepareStatement("SELECT * FROM USERS WHERE PASSWORD=?"); prep.setString(1, pwd);`

## SQL Injection countermeasures: SQL stored procedures

- Using stored procedures (system or user-defined) per se does not guarantee against SQL injection if they do not validate the user input
- Moreover, stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows

## SQL Injection countermeasures summary

- Constrain and sanitize input data. Check for known good data by validating for type, length, format, and range.
- Use type-safe SQL parameters for data access. You can use these parameters with stored procedures or dynamically constructed SQL command strings.
- Use an account that has restricted permissions in the database. Ideally, you should only grant execute permissions to selected stored procedures in the database and provide no direct table access.
- Avoid disclosing database error information. In the event of database errors, make sure you do not disclose detailed error messages to the user.
- Encrypt Sensitive Data stored in Database
- Harden DB Server and Host O/S

## Application security

- If the access control to the data is implemented by the application (for example an application accessible from the Web), then the following checklist of possible threats to the application should be also considered

## Application security profile (1)

Set of questions to ask (and their respective answers) , organized by vulnerability category

Category	Considerations
Input validation	<ul style="list-style-type: none"> <li>Is all input data validated?</li> <li>Could an attacker inject commands or malicious data into the application?</li> <li>Is data validated as it is passed between separate trust boundaries (by the recipient entry point)?</li> <li>Can data in the database be trusted?</li> </ul>
Authentication	<ul style="list-style-type: none"> <li>Are credentials secured if they are passed over the network?</li> <li>Are strong account policies used?</li> <li>Are strong passwords enforced?</li> <li>Are you using certificates?</li> <li>Are password verifiers (using one-way hashes) used for user passwords?</li> </ul>
Authorization	<ul style="list-style-type: none"> <li>What gatekeepers are used at the entry points of the application?</li> <li>How is authorization enforced at the database?</li> <li>Is a defense in depth strategy used?</li> <li>Do you fail securely and only allow access upon successful confirmation of credentials?</li> </ul>
Configuration management	<ul style="list-style-type: none"> <li>What administration interfaces does the application support?</li> <li>How are they secured?</li> <li>How is remote administration secured?</li> <li>What configuration stores are used and how are they secured?</li> </ul>

## Application security profile (2)

Set of questions to ask (and their respective answers) , organized by vulnerability category

Category	Considerations
Session management	<ul style="list-style-type: none"> <li>How are session cookies generated?</li> <li>How are they secured to prevent session hijacking?</li> <li>How is persistent session state secured?</li> <li>How is session state secured as it crosses the network?</li> <li>How does the application authenticate with the session store?</li> <li>Are credentials passed over the wire and are they maintained by the application? If so, how are they secured?</li> </ul>
Cryptography	<ul style="list-style-type: none"> <li>What algorithms and cryptographic techniques are used?</li> <li>How long are encryption keys and how are they secured?</li> <li>Does the application put its own encryption into action?</li> <li>How often are keys rotated?</li> </ul>
Parameter manipulation	<ul style="list-style-type: none"> <li>Does the application detect tampered parameters?</li> <li>Does it validate all parameters in form fields, view state, cookie data, and HTTP headers?</li> </ul>
Exception management	<ul style="list-style-type: none"> <li>How does the application handle error conditions?</li> <li>Are exceptions ever allowed to propagate back to the client?</li> <li>Are generic error messages that do not contain exploitable information used?</li> </ul>
Auditing and logging	<ul style="list-style-type: none"> <li>Does your application audit activity across all tiers on all servers?</li> <li>How are log files secured?</li> </ul>

## SQL injection resources

- Advanced SQL Injection In SQL Server Applications. An NGSSoftware Insight Security Research (NISR) Publication ©2002 Next Generation Security Software Ltd [http://www.nextgenss.com/papers/advanced\\_sql\\_injection.pdf](http://www.nextgenss.com/papers/advanced_sql_injection.pdf)
  - discusses in detail the common 'SQL injection' technique, as it applies to the popular Microsoft Internet Information Server/Active Server Pages/SQL Server platform.
- SQL Injection Walkthrough (the hacker point of view) <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- CAPEC SQL Injection attack (Attack Pattern ID 66) <http://capec.mitre.org/data/definitions/66.html>
  - Detailed description of SQL Injection attack, solutions and mitigations
- William G.J. Halfond, Jeremy Viegas, and Alessandro Orso A Classification of SQL Injection Attacks and Countermeasures In Proc. of the Intern. Symposium on Secure Software Engineering (ISSE 2006), Mar. 2006 <http://www.cc.gatech.edu/~whalfond/papers/halfond06isssse.pdf>