Recap

**Injection attacks** use specially crafted inputs to subvert the intended operation of applications.

- **OS Command Injections** may execute arbitrary commands.
- **SQL Injections** can reveal database contents, affect the results of queries used for authentication; sometimes they can even execute commands.

In this lecture we look at **SQL Injections** in some detail.

Context

SQL Injection (SQLi) is by some estimates the current **number one** category of software vulnerability.

As with overflows, there is a **large body of crafty exploits** made possible by (often small) errors in coding or design.

We will look at:

- SQLi attack types and mechanisms
- detecting SQLi
- preventing SQLi

Even if you believe you are safe from SQLi, it is useful to understand the range of problems and solutions. And a “NoSQL” database doesn’t mean no SQL-like injections!

SQL Queries

SQL: standard language for interacting with databases

- very common with **web applications**
  - authentication: DB of users, passwords
  - main function often data storage
- but also in **desktop and server** apps
  - email clients/servers
  - photo applications, media servers
  - custom database clients
  - application data caches

**Question.** Why might the second category cause concern for security auditing?
Network versus local injections

**Network** usually considered the bigger risk
- Access by many, unknown users
- Network is gateway, crossing physical boundaries
- Risk in privileged servers (setguid, etc)

**Local** inputs: should they be considered too?
- Local users can only deny access to themselves
- Desktop apps run as plain user, only risk own data

However, this trust assumption can be wrong:
- *drive-by exploits* attack locally (or use escalation)
- Growing concerns over insider threats


__Advisory RFP2K01 —————————— rfp.labs ————

"How I hacked PacketStorm"
A look at hacking wwwthreads via SQL

______________ rain forest puppy / rfp@wiretrip.net —___

- One of the first public examples and explanation
- Demonstrated retrieval of 800 passwords
- See Rain Forest Puppy’s advisory and his earlier Phrack 54 article

Man steals 130m card records (2009)

Provocation: Swedish pencil vote (2010)

Did Little Bobby Tables migrate to Sweden?

Posted by Jonas Eklöf Thu, 23 Sep 2010 20:30:00 GMT

As you may have heard, we’ve had a very close election here in Sweden. Today the Swedish Election Authority published the [hand written votes](http://www.valstiftelsen.se). While scanning through them I happened to notice

R:13: Hallands län:88; Haltedal:81; Halsteda västra valkrets, 9904:500r=4, 500 DROP TABLE VAL;1

The second to last field is the actual text on the ballots. Could it be that Little Bobby Tables is all grew up and has migrated to Sweden? Well, it’s probably just a pike but even so it brings questions since an [SQL injection](http://en.wikipedia.org/wiki/SQL_injection) on election data would be very serious.

Someone even tried to get some JavaScript in there:


src=http://ftpas.web.com/x:500:1

I’m pleased to see that they published the list as text and not HTML. This

Should know better (2011)

The A Register

Under the microscope: The bug that caught PayPal with its pants down
Payment giant suffers textbook SQL injection flaw

By John Leyden

The first superbug

Security researchers have published an [extraordinary](http://www.securityfocus.com/%E2%80%9C%EF%BF%8B) new case of a recently patched SQL injection flaw in PayPal’s version

42
Should know better (2015)

TalkTalk: MPs to hold inquiry into cyber-attack

TalkTalk

Provocation: British company name (2016)

; DROP TABLE "COMPANIES";-- LTD

Company number 10542919

Registered office address
1 Moyes Cottages Bentley Hall Road, Capel St. Mary, Ipswich, Suffolk, United Kingdom, IP8 2LA.

Company status
Active

Company type
Private limited company

Incorporated on 29 December 2016

See the recent reddit thread

Typical setting for attacks

Picture from SQL Injection Attacks and Defense, J. Clarke, Syngress, 2012

Typical vulnerability in PHP code

$username = $HTTP_POST_VARS['username'];
$password = $HTTP_POST_VARS['passwd'];

$query = 'SELECT * FROM logintable WHERE user = ''
. $username . '' AND pass = '' . $password . '';

$result = mysql_query($query);

if (!$results)
die_bad_login();

Guaranteed login! Try with:
user name: bob' OR user='bob
password: foo OR pass='foo

which gives

SELECT * FROM logintable WHERE user=
'bob' or user='bob' AND pass='foo' OR pass='foo'

Fixes: in-band versus out-of-band

▶ The “in-band” solution is to use filtering to escape black-listed characters.
▶ PHP and MySQL provide functions to help do this, guaranteeing meta-characters are quoted.
▶ The “out-of-band” fix is to use a prepared query with parameters carved out for the substituted positions.
▶ Prepared query has placeholders for parameters which will be safely substituted.

Question. Why might the out-of-band fix be preferable?
An example in Java servlet code

```java
public class Show extends HttpServlet {
    public ResultSet getuserInfo(String login, String pin) {
        Connection conn = DriverManager.getConnection("MyDB");
        Statement stmt = conn.createStatement();
        String queryString = "SELECT accounts FROM users WHERE ";
        if (!login.equals("")) && (!pin.equals(""))) {
            queryString += "login='" + login + " AND pin='" + pin; 
        } else {
            queryString+="login='guest"; 
        }
        ResultSet tempSet = stmt.execute(queryString);
        return tempSet;
    }
}
```

Normal usage

```java
queryString = "SELECT accounts FROM users WHERE ";
if (!login.equals("")) && (!pin.equals(""))) {
    queryString += "login='" + login + " AND pin='" + pin; 
} else {
    queryString+="login='guest"; 
}
```

User submits login="john" and pin="1234"
SQL issued:
```
SELECT accounts FROM users WHERE login='john' AND pin=1234
```

Malicious usage

```java
queryString = "SELECT info FROM users WHERE ";
if (!login.equals("")) && (!pin.equals(""))) {
    queryString += "login='" + login + " AND pin='" + pin; 
} else {
    queryString+="login='guest"; 
}
```

User submits login="admin' --" and pin="0"
SQL issued:
```
SELECT accounts FROM users WHERE login='admin' --' AND pin=0
```

Quotation and meta-characters

The warnings about meta-characters in shell commands apply equally to SQL. And they can vary according to the underlying DB engine, and flags which configure it...

Classifying SQL injections

There are a wide variety of SQL injection techniques. Sometimes several are used to mount a single attack.
It’s useful to examine:

- **route** — where injection happens
- **motive** — what it aims to achieve
- **SQL code** — the form of SQL injected

Injection routes

- **User input** e.g., web forms via HTTP GET or POST
- **Cookies** used by web apps to build queries
- **Server variables** logged by web apps (e.g., http headers)
- In so-called **second-order injections** the injection is separated from attack

These slides follow A Classification of SQL Injection Attacks and Countermeasures by Halfond, Viegas and Orso. ISSE 2006.
Primary and auxiliary motives

**Primary** motives may be:
- Extracting data
- Adding or modifying data
- Mounting a denial of service attack
- Bypassing authentication
- Executing arbitrary commands

**Auxiliary** motives may be:
- Finding injectable parameters
- Database server fingerprinting
- Finding database schema
- Escalating privilege at the database level

Forms of SQL code injected

1. **Tautologies**
2. Illegal/incorrect queries
3. Union query
4. Piggy-backed queries
5. Inference pairs
6. Stored procedures and other DBMS features

Additionally, the injection may use *alternate encodings* to try to defeat sanitization routines that don’t interpret them (e.g., char(120) instead of x).

**Exercise.** For each of these types (described next), consider which primary/secondary motive(s) might apply.

Tautologies

Inject code into condition statement(s) so they always evaluate to true.

```
SELECT accounts FROM users WHERE
login=' ' OR 1=1 -- AND pin=
```

Blacklisting tautologies is difficult

- Many ways of writing them: l>0, 'x' LIKE 'x', ...
- **Quasi tautologies**: very often true RAND() >0.01, ...

**Question.** Instead of a tautology, can you think of how an attacker might use an always-false condition?

Illegal/incorrect

Cause a run-time error, hoping to learn information from error responses.

```
SELECT accounts FROM users WHERE
login=' ' AND pin=convert(int,(select top 1 name from sysobjects where xtype='u'))
```

- Supposes MS SQL server
  - sysobjects is server table of metadata
- Tries to find first user table
- Converts name into integer: runtime error

**Example response**

Microsoft OLE DB Provider for SQL Server (0x80040E07)
Error converting nvarchar value 'CreditCards' to a column of data type int

Tells the attacker:
- MS SQL Server is running
- The first user-defined table is called CreditCards

Union query

Inject a second query using UNION:

```
SELECT accounts FROM users WHERE
login=' ' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND pin=
```

- Suppose there are no tuples with login=''
- Result: may reveal cardNo for account 10032
**Piggy-backed (sequenced) queries**

The Bobby Tables attack is an example of a piggy-backed query.

The attacker injects a second, distinct query:

```
SELECT accounts FROM users WHERE login='doe'; drop table users --
```

- Database parses second command after ';'
- Executes second query, deleting `users` table
- NB: some servers don’t need ; character

**Inference pairs**

Suppose error responses are correctly captured and not seen by the client.

It might still be possible to extract information from the database, by finding some difference between outputs from pairs of queries.

- **A Blind Injection** tries to reveal information by exploiting some visible difference in outputs.
- **A Timing Attack** tries to reveal information by making a difference in response time dependent on a boolean (e.g., via \texttt{WAITFOR})

If the attacker has unlimited access, these can be used in repeated, automated, differential analysis.

**Blind injection example**

Idea: discover whether login parameter is vulnerable with two tests.

**Step 1.** Always true:

```
SELECT accounts FROM users WHERE login='legalUser' and 1=1 -- '
```

RESPONSE: INVALID PASSWORD

The attacker thinks:

- Perhaps my invalid input was detected and rejected, or perhaps the username query was executed separately from the password check.

**Step 2.** Always false:

```
SELECT accounts FROM users WHERE login='legalUser' and 1=0 -- '
```

RESPONSE: INVALID USERNAME AND PASSWORD

The attacker thinks:

- Aha, the response is different! Now I can infer that the login parameter is injectable.

**Stored procedures**

**Stored procedures** are custom sub-routines which provide support for additional operations.

- May be written in scripting languages.
- Can open up additional vulnerabilities.

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pin int
AS
EXEC ("SELECT accounts FROM users WHERE login='" + @userName + "' and pass='" + @pass + "' and pin=" + @pin + ");
GO
```

\texttt{varchar2} is an Oracle datatype for variable length strings.
Stored procedures

This is invoked with something like:

```sql
EXEC DBO.isAuthenticated 'david' 'bananas' 1234
```

Or something like:

```sql
EXEC DBO.isAuthenticated ' ; SHUTDOWN; --' '' ''
```

which results in:

```sql
SELECT accounts FROM users WHERE
login='doe' pass=' '; SHUTDOWN; -- AND pin=
```

An especially dangerous stored procedure

Microsoft SQL Server offers: `xp_cmdshell`, which allows operating system commands to be executed!

```sql
EXEC master..xp_cmdshell 'format c:'
```

▶ Since SQL Server 2005, this is disabled by default
▶ . . . but can be switched back on by DB admins
▶ . . . maybe from inside the db?!

Lesson: access control and passwords may be critical inside the DB, even for restricting attacks outside.

Other database server features

There are other features offered variously depending on the DBMS.

For example, queries in MySQL can write files with the idiom:

```sql
SELECT INTO outfile.txt ...
```

Question. Why might writing files be of use to an attacker?

How do I repair an SQLi vulnerability?

Mentioned earlier:

- filtering to sanitize inputs
- prepared (aka parameterized) queries

Both methods are server side, so it is better to use database driver libraries to abstract away from the underlying DBMS.

In Java, JDBC provides the `PreparedStatement` class.

We'll look at further relevant secure coding issues later lectures; in particular, ways of managing input and also output filtering.

Question. What type of SQL attacks might `PreparedStatement` not prevent against?

How do I prevent SQLi vulnerabilities?

Choice of stages (as usual):

1. eliminate before deployment:
   - manual code review
   - automatic static analysis
2. in testing or deployment:
   - pen testing tools
   - instrumented code
3. after deployment:
   - wait until attacked, manually investigate
   - use dynamic remediation plus alarms (app firewall or specialised technique)

Some examples follow.
Detection externally: pen testing tools

These incorporate the injection methods shown before, to explore a server for known vulnerabilities.

Static prevention: automated analysis

Idea: static code analysis used to warn programmer or prohibit/fix vulnerable code.
Techniques:
▶ Detect suspicious code patterns, e.g., dynamic query construction
▶ Use static taint analysis to detect data-flows from input parameters to queries

We’ll look at static analysis in more detail in later lectures

Dynamic detection tool: AMNESIA

Idea: use static analysis pre-processing to generate a dynamic detection tool:
1. Find SQL query-generation points in code
2. Build SQL-query model as NDFA which models SQL grammar, transition labels are tokens
3. Instrument application to call runtime monitor
4. If monitor detects violation of state machine, triggers error, preventing SQL query

State machine for SQL production

▶ Variable beta: matches any string in SQL grammar
▶ Spots violation in injectable parameters
   ◦ abort query if model not in accepting state

See Halfond and Orso, AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, Automated Software Engineering, 2005

Dynamic prevention: SQLrand

Idea: use instruction set randomization to change language dynamically to use opcodes/keywords that attacker can’t easily guess.

Review questions

SQLi classification
▶ Describe three routes for SQL injection.
▶ Describe three auxiliary motives that an attacker may have when using SQL injection techniques to learn about a target.

SQLi prevention and detection
▶ How would you repair the prototypical example SQLi vulnerability?
▶ Describe automatic ways to prevent and detect SQLi vulnerabilities.

See Boyd and Keromytis, SQLrand: Preventing SQL Injection Attacks, Applied Cryptography and Network Security, 2004
References and credits

This lecture includes content adapted from:

- *A Classification of SQL Injection Attacks and Countermeasures* by Halfond, Viegas and Orso. ISSE 2006