Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Advanced Web Technology
9) Injection Flows

Emmanuel Benoist
Fall Term 2016-17

Computer Science Division

Table of Contents

Presentation

Injection Flows

- **Principle:**
  - Occurs when user supplied data is sent to an interpreter as part of a command or a query.
- **Injection Flows may be done on:**
  - SQL (most common)
  - LDAP
  - XPath
  - XSLT
  - HTML
  - OS Command injection
  - ...
- **This vulnerability is very common on Web Application**

How does it work?

- **Attacker tricks the interpreter into executing unintended command**
- **Attacker supplies unexpected content to a site**
  - Data is especially designed to fool the site
- **Attacker may take control of the interpreter, for instance SQL:**
  - Read data (unintended, of course)
  - update, delete or create any arbitrary data
- **For the Operating System interpreter**
  - Attacker may have the opportunity to execute any command

Vulnerability

Vulnerability

- **Environments affected**
  - Any framework using an interpreter or invoke process
  - SQL
  - Command line
  - ...
- **System is vulnerable when user input is passed without tests**

PHP

```
$query = "select * from guestbook";
$query .= " where  title like '".$_REQUEST['search'];
$result = mysql_query($query , $conn);
```

Java

```
String query = "select * from user where username='";
query += req.getParameter("userID");
query += "' and password = '"+req.getParameter("pwd")↘
    +"'";
```


Protection

Protection

- **Avoid the use of interpreter if possible**
- **Otherwise: Use safe APIs**
  - Strongly typed parameterized queries
  - Object Relational Mapping (ORM)

  They handle data escaping
- **Validation is still recommended**
  - in order to detect attacks

Take Extra Care when using interpreters

- **Input Validation**
  - Validate all input data: length, type, syntax, business rules
  - validation is done before displaying or storing any data
  - Validation must be done server-side
  - Javascript validation doesn't bring any security
- **Use strongly typed parameterized query APIs**
  - with placeholder substitution markers,
- **Enforce least privilege**
  - Configure your DB such that the web account can't do more than what is expected
  - restrict the rights of your user when executing an OS command

Take Extra Care (Cont.)

- **Avoid detailed error messages**
  - Give access to versions numbers
  - Give access to parts of the code
  - Give access to configurations
- **Use stored procedures**
  - They are generally safe from SQL injection
  - Can however be injected (for instance using `exec()`)
- **Do not use dynamic query interfaces (such as `mysql_query()`)**

Take Extra Care (Cont.)

- **Do not use simple escaping functions**
  such as
  - `addslashes()` in PHP
  - `str_replace("'","''")`
  - it is weak and has been successfully exploited
- **Prefer following methods**
  - use `mysql_real_escape_string()`
  - or preferably PDO which does not require escaping

Language Specific recommendations

- **Java EE**
  - use strongly typed `PreparedStatement`
  - or use an ORM (Object Relational Manager) such as Hibernate or Spring
- **PHP**
  - Use PDO with strongly typed parameterized queries (using `bindParam()`).

Examples

Which site is subject to SQL injection?

- **Such a site must access a DB**
  - The parameter should be given by the user
  - This parameter is then used to select data in the DB
  - Example `www.mysite.com/index.php?id=100`
  - Means there exists a request for the page number 100
- **If the site does not test its input**
  - You can test it by typing something like: `www.mysite.com/index.php?id=%2710`
- **If the site lets the user see error messages**
  - Test the output of your input
- **Examples**
  - Search form (SELECT with LIKE)
  - Login form (SELECT with two =)
  - Insertion of new entries
  - ...

Example: Presentation

- **Suppose we have the following HTML Form**

```
<form method="POST">
    <input type="text" name="username"><br>
    <input type="password" name="password"><br>
    <input type="submit" value="Login">
</form>
```

- **and the following PHP line defining a SQL command:**

```
$query = "SELECT * FROM user WHERE user='$user'";
$query .= " AND password='$pwd'";
```

- **For our examples, we disable `magic_quotes_gpc` `php.ini` file** (if this option is on, and it quotes all GET, POST and COOKIES parameters, means chars like: " and ' are escaped and become \" and \')

  magic_quotes_gpc = off

Example: Select user and password

We want the select to work in any case

- **Following expressions are always true**

  SELECT * FROM table WHERE 1=1;
  SELECT * FROM table WHERE 1;
  SELECT * FROM table WHERE ISNULL(NULL)
  SELECT * FROM table WHERE 1 IS NOT NULL
  SELECT * FROM table WHERE NULL IS NULL
  ...

- **So we do not need a valid username and password**
  if `$user="' OR 'a'='a"` and $password remains empty then the previous expression becomes:

  SELECT * FROM user WHERE user='' OR 'a'='a' AND ↘
  →password='';

  - Returns the list of all the users
  - So we are logged in with the first provided

Login on a specific account

- **We can specify the right username and change the password**
  - If we give `$user="Emmanuel"`
  - And `$password="' OR 'b' BETWEEN 'a' AND 'c"`
- **The previous SQL statement becomes**

  SELECT * FROM user WHERE user='Emmanuel' AND ↘
  →password='' OR 'b' BETWEEN 'a' AND 'c';

- **So username is OK, but password is not checked!**

Example, using Comments

- **Another great principle in SQL injection is Comments**
  It is also very common in all the other injections
- **If we inject a #, the rest of the SQL expression is not evaluated**
  if `$user="John' #"` the request becomes

SELECT * FROM user WHERE username='John'*#' AND* ↘
→*password=''*

which is equal to

SELECT * FROM user WHERE username='John'

- **If we use the comments `/* comments */` we may escape some tests**

More injection in SELECT

- **Suppose we have the following query, for displaying the content of one single comment in our guestbook:**

  $query = "select␣*␣from␣guestbook␣where␣guestbookID=$nb";

- **We can copy the content in a file**
  - suppose we define
    `$number="11 or 1=1 INTO OUTFILE '/tmp/test.security.txt'"`
  - The total content of the table is sent to a file.
- **Suppose the Attacker has an account on the system (e.g. `foobar`).**
  - It is possible to change the password of `foobar`
  - **Attacker could create any php file inside the system!!**

SELECT password FROM user WHERE login='foobar' INTO ↘
→OUTFILE '/opt/lampp/htdocs/test.php'

Example: INSERT INTO

- **Attacker can also manipulate INSERT INTO queries**
  - Use the knowledge of the DB to input unsolicited data
- **Example: Suppose we have a table `user`:**
  - `userID` (auto-increment), `username`, `password`, `email` `userlevel`
- **We have a register procedure containing following query**

$query = "INSERT␣INTO␣user␣(username,password,email,↘
→userlevel)";
$query .= "␣VALUES␣('$username','$password','$email','1')"

- **Suppose we give the value `$email="',3)#"`**
  - Element is inserted with privilege "3" (= admin) whereas he should be only "1" (= user).

Example: UPDATE

- **Suppose we have the possibility to change the password of one user**

  UPDATE 'user' SET 'password' = '$pwd1' WHERE 'userID'↘
  → ='$uid' LIMIT 1;

- **We could also change the level of the user**

  $pwd1 = "mypwd',␣userlevel='3";

  Which creates the following request:

  UPDATE 'user' SET 'password' = 'mypwd', userlevel='3' ↘
  →WHERE 'userID' ='$uid' LIMIT 1;

Example: UPDATE (Cont.)

- **Or change the password of any other user**

  $pwd1 = "mypwd'␣WHERE␣userID␣=␣10#";

  Which creates the following request:

  UPDATE 'user' SET 'password' = 'mypwd' WHERE userID ↘
  →= 10

Example UPDATE (Cont.)

- **Suppose we have a table for news. Visitors can give a note to each news.**
  - We have the following table `news`:
    `newsID`, `title`, `content`, `votes` (number), `score` (number)
  - The following query is used to count one vote:

    UPDATE news SET votes=votes+1, score=score+$note ↘
    →WHERE newsID='$id'

  - We have the following attack

    $note="3,␣title='hop'

- **Why is this interesting?**
  - Attacking numbers doesn't require ' or "
  - Is compatible with `magic_quotes_gpc = on`

Example : `UNION ALL`

- `UNION ALL` **is used to concatenate two queries**
  - Written at the end of a select query, concatenates the two results

  select name, price from article where price>10 union all ↘ →select username, password from user;

- **Taken as one result set in programming languages**
  - `UNION ALL` is transparent for the program
  - works exactly as if the select was normal
  - The two selects need to have the same number of columns
- **Example in the Guestbook**
  - Insert this instructions inside the search area
  - Done in Exercise

Attacks using no quotes

- **Since most of the server have** `magic_quotes_gpc = on`
  - Attackers can not use ' or "
- **Use MySQL char() function**
  - Returns the character denoted by the number,
  - For instance `char(104,111,112)` returns the string `hop`
- **Previous attack becomes**
  - The following query is used to count one vote:

    UPDATE news SET votes=votes+1, score=score+$note ↘ →WHERE newsID='$id'

    ,
  - We have the following attack

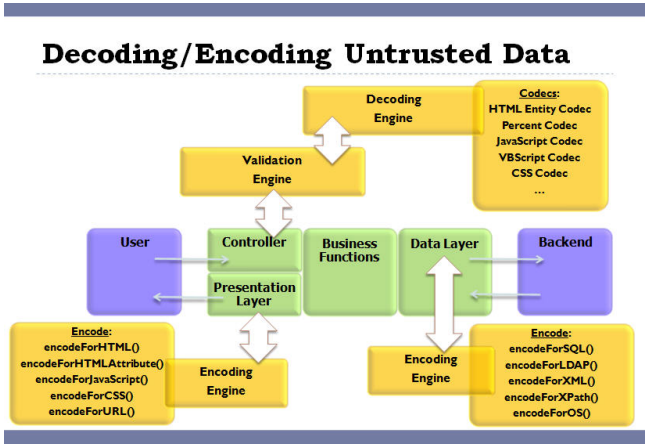
    $note="3,_title=char(104,111,112)

How to protect yourself
From SQL injection in PHP

- **Configure PHP such that ' and " are automatically escaped (not satisfactory)**

  magic_quotes_gpc = on
- **Always quote input before sending query to an interpreter (not satisfactory)**
  - `mysql_real_escape_string()`
- **Much Better: Do not use any interpreter at all**
  - Use PDO

Decoding / Encoding Untrusted Data[1]



[1]Source: Javadoc documentation of the ESAPI package

Shell Injection

Shell Injection[2]

- **Shell Injection is named after Unix shells,**
- **But it applies to most systems which allows software to programmatically execute command line.**
- **Typical sources of Shell Injection is calls:**
  - `system()`,
  - `StartProcess()`,
  - `java.lang.Runtime.exec()`,
  - `System.Diagnostics.Process.Start()`
  - and similar APIs.
- **Considere the following short program**

```
<?php
passthru ( "␣/home/user/phpguru/funnytext␣"
           . $_GET['USER_INPUT'] );
?>
```

[2] Source: Wikipedia

Shell Injection (Cont.)

This program can be injected in multiple ways:

- `‘command‘` **will execute command.**
- `$(command)` **will execute command.**
- `; command` **will execute command, and output result of command.**
- `| command` **will execute command, and output result of command.**
- `&& command` **will execute command, and output result of command.**
- `|| command` **will execute command, and output result of command.**
- `> /home/user/phpguru/.bashrc` **will overwrite file .bashrc.**
- `< /home/user/phpguru/.bashrc` **will send file .bashrc as input to funnytext.**

Examples of injection

Suppose we have the following shell

```
<?php
if(isset($_GET['name'])){
  system('echo␣'.$_GET['name']);
}
?>
```

The following content will hack the system

- `‘ls ../../..‘` Executes a command, the returned value is given as a parameter to `echo`.
- Produces the following command line:

  echo ‘ls ../../..‘
- `$(cat /home/bie1/.emacs)` Displays the content of the emacs config file of user bie1.

  echo $(cat /home/bie1/.emacs)

Examples of injection (Cont.)

- `; touch /tmp/myfile.txt` Creates the following command

  echo ; touch /tmp/myfile.txt

  Makes a `echo`, then starts something new, it creates a new file `/tmp/myfile.txt` which is empty.
- `Hello World | wc` creates the following command line:

  echo Hello World | wc

  It makes a `echo` then its output is transfered to the `wc` (word count).
- `test > /tmp/test2.txt` Creates:

  echo test > /tmp/test2.txt

  It writes in the file `/tmp/test2.txt` the content that is given as output by `echo`.

Attacks using shell injection flow

- **An attacker can create any type of file**
  - A txt file
  - A PHP file
  - A shell file
- **Can see and modify config files**
  - Can visit directories
  - Can `cat` the content of a file
  - Can overwrite the content of an existing file
- **Attacker inherits the strength of web user**
  - If web server is run as a normal user: lot of possibilities
  - If the web user is restricted to the minimum, risk is smaller.

Defense agains Shell Injection

- **PHP offers functions to perform encoding before calling methods.**
  - `escapeshellarg()`
  - and `escapeshellcmd()`
- **However, it is not recommended to trust these methods to be secure**
- **also validate/sanitize input.**

XML-Injection

XML-Injection[3]

- **The attacker trys to inject XML**
  - The application relies on XML (stores information in an XML DB for instance)
  - The information provided by the attacker is evaluated together with the existing one.
- **We will see a practical example**
  - A XML style communication will be defined
  - Method for inserting XML metacharacters
  - Then the attacker has information about the XML structure
  - Possibility to inject XML data and tags.

[3]Source: OWASP Testing Guide

Black Box testing

Example

- **Let us suppose we have the following xmlDB file (information is stored in an XML)**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
</users>
```


Insertion of a new user

- **Is done with a form (with the GET method)**
  - Three fields: `username`, `password` and `email`
- **Suppose the clients sends the following values**
  - username=Emmanuel
  - password=B3n0is7
  - email= emmanuel@bfh.ch
- **It produces the following GET request**

http://*www.benoist.ch/addUser.php?username=Emmanuel&*
password=B3n0is7&email=emmanuel@bfh.ch

Insertion of a new user (Cont.)

- **The program will create a new XML `user`-node**

```
<user>
  <username>Emmanuel</username>
  <password>B3n0is7</password>
  <userid>500</userid>
  <mail>emmanuel@bfh.ch</mail>
</user>
```

- **The new entry in entered inside the XML DataBase**

Conclusion

Conclusion

- **SQL Injection allows attacker to**
  - Read data: Access passwords, data stored
  - Change Data : Access security level
  - Delete data
  -
- **SQL injection Vulnerabilities opens the door to:**
  - Privacy breach : Data can be accessed without consent
  - Identity theft : idem + failure in authentication
  - Compromission of the system : write of new files (maybe PHP)
  - ...
- **Easy protection are already exploited**
  - Adding one (or more) layers between presentation and database layer is a must (also from the point of view of Design)
  - Even this has also been successfully exploited.
- **Solution? test your inputs!**

Conclusion

Conclusion (Cont.)

- **Shell Injection**
  - Attacker inherits the priviledges of the user running the web server
  - Solutions: Filter/Sanitize input + reduce the priviledges to the minimum
- **XML Injection**
  - Attacker can force the server to load entities from outside
  - He can change the content of an XML database, and gain illegal priviledges in the application.
  - Solution: Filter/Sanitize input, allow no metacharcters in your normal inputs, or escape them.

References

- **OWASP Top 10 - 2007**
  `http://www.owasp.org/index.php/Top_10_2007`
- **A Guide for Building Secure Web Applications and Web Services**
  `http://www.lulu.com/content/1401012`
- Advanced SQL Injection in SQL Server Applications - Chris Anley
  `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`
- L'injection (My)SQL via PHP - leseulfrog
  `http://www.phpsecure.info/v2/article/InjSql.php`
  Advanced version:
  `http://www.phpsecure.info/v2/article/phpmysql.php`
- SQLMAP (a SQL Injection Tool)
  `http://sqlmap.sourceforge.net`