Injection

Injection

2021 OWASP Top 10	
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Component
7	Identification and Authentication Failu
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failu
10	Server-Side Request Forgery (SSRF)

2021 CWE Top 25 (MITRE)	
1	Out-of-bounds Write
2	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	Out-of-bounds Read
4	Improper Input Validation
5	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7	Use After Free
8	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	Cross-Site Request Forgery (CSRF)
10	Unrestricted Upload of File with Dangerous Type
11	Missing Authentication for Critical Function
12	Integer Overflow or Wraparound
13	Deserialization of Untrusted Data
14	Improper Authentication
15	NULL Pointer Dereference
16	Use of Hard-coded Credentials
17	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	Missing Authorization
19	Incorrect Default Permissions
20	Exposure of Sensitive Information to an Unauthorized Actor
21	Insufficiently Protected Credentials
22	Incorrect Permission Assignment for Critical Resource
23	Improper Restriction of XML External Entity Reference
24	Server-Side Request Forgery (SSRF)
25	Improper Neutralization of Special Elements used in a Command ('Command Injection')

mgm

SQL Injection

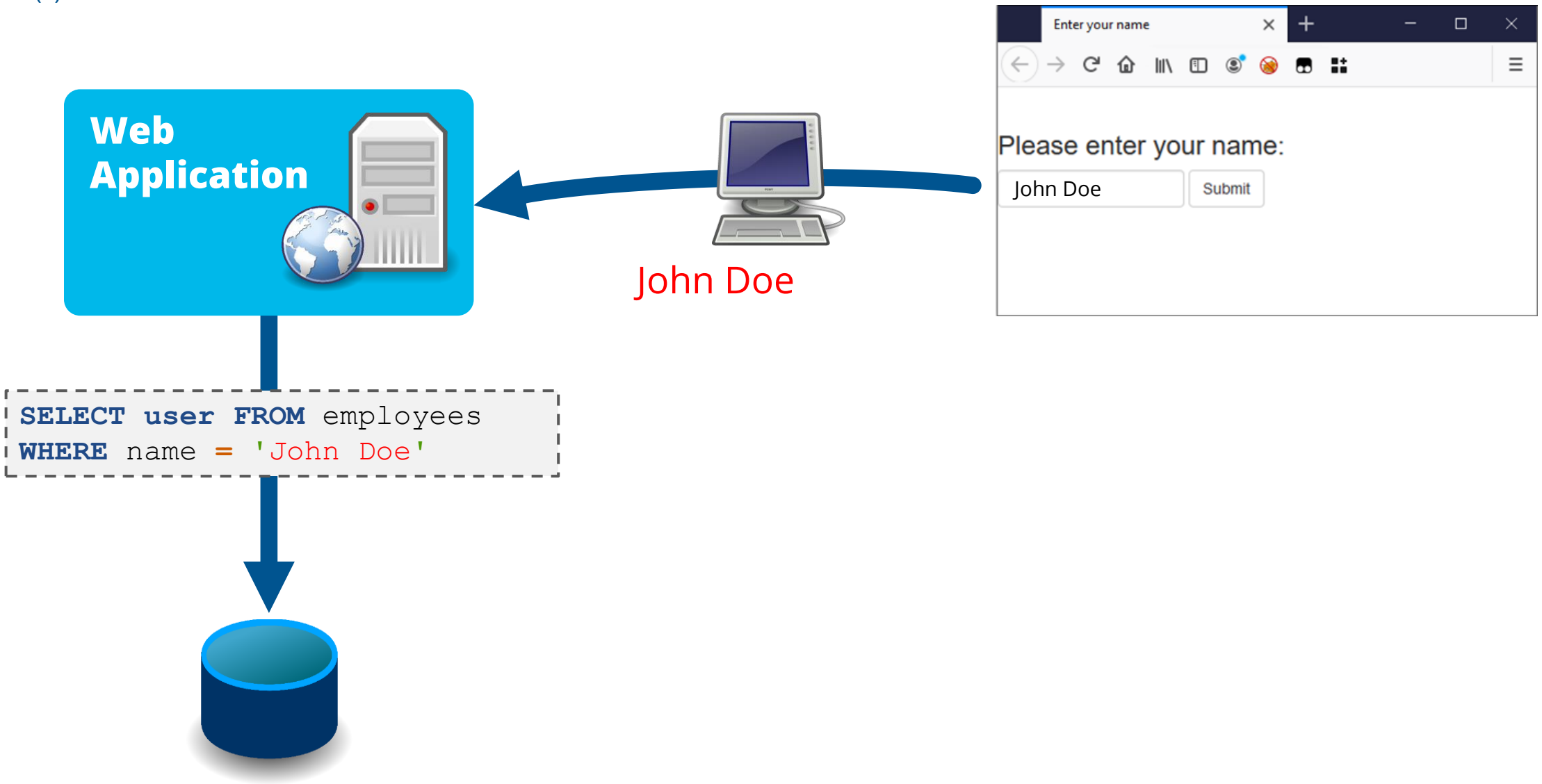
2021 OWASP Top 10	
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery (SSRF)

2021 CWE Top 25 (MITRE)	
1	Out-of-bounds Write
2	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	Out-of-bounds Read
4	Improper Input Validation
5	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injecti
6	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7	Use After Free
8	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	Cross-Site Request Forgery (CSRF)
10	Unrestricted Upload of File with Dangerous Type
11	Missing Authentication for Critical Function
12	Integer Overflow or Wraparound
13	Deserialization of Untrusted Data
14	Improper Authentication
15	NULL Pointer Dereference
16	Use of Hard-coded Credentials
17	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	Missing Authorization
19	Incorrect Default Permissions
20	Exposure of Sensitive Information to an Unauthorized Actor
21	Insufficiently Protected Credentials
22	Incorrect Permission Assignment for Critical Resource
23	Improper Restriction of XML External Entity Reference
24	Server-Side Request Forgery (SSRF)
25	Improper Neutralization of Special Elements used in a Command ('Command Injection')

mgm

The Principle

SQL Injection (1)



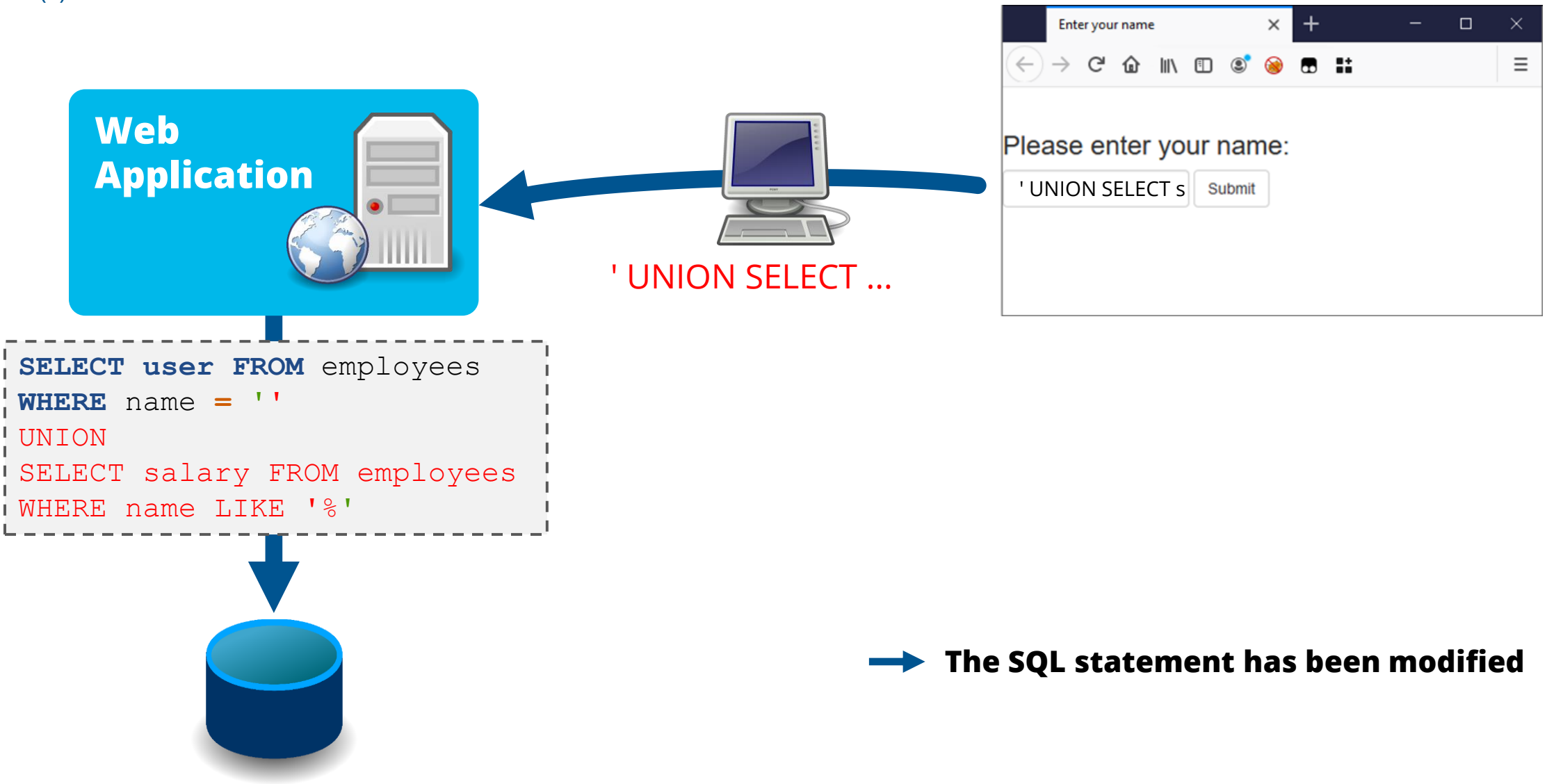
The Principle

SQL Injection (2)

```
1  Connection connection = /* some DB connection */
2  Statement statement = connection.createStatement()
3
4  String queryString = "SELECT user FROM employees "
5          + "WHERE name = '"+ request.getParameter("fullname") + "'";
6
7  ResultSet results = statement.executeQuery(queryString);
```

The Principle

SQL Injection (3)



→ **The SQL statement has been modified**

SQL Injection (“classic”)

Concatenated Query in the Application Server (**Bad!**)

```
1  Connection connection = /* some DB connection */
2  Statement statement = connection.createStatement()
3
4  String queryString = "SELECT user FROM employees "
5      + "WHERE name = '" + request.getParameter("fullname") + "'";
6
7  ResultSet results = statement.executeQuery(queryString);
```

```
SELECT user FROM employees WHERE name = '_____________';
```

```
' OR 'a' = 'a
```

```
SELECT user FROM employees WHERE name = '' OR 'a' = 'a';
```

Example

SQL Injection - Consequences

- Several attacks can be conducted:

```
UNION SELECT balance FROM account;

UPDATE interest SET ...

DELETE ...;

INSERT ...;
```

- and access to the file system:

```
CREATE TABLE footable(data longblob);                                    -- create BLOB table
INSERT INTO footable(data) VALUES(0x4d5a90…610000);                      -- fill table with binary
UPDATE footable SET data = CONCAT(data, 0xaa270000…000000);              -- data
[…];
SELECT data FROM footable INTO DUMPFILE 'C:/WINDOWS/Temp/nc.exe'; -- drop finished malware
```

One vulnerable web application

may compromise the security of the whole system

!

SQL Injection

Fun with SQL Syntax

```
1 OR 1=1

1 OR (1)=(1)

1 OR (DROP TABLE users)=(1)

CONCAT(CHAR(39),CHAR(07),CHAR(39))

1 OR ASCII(2) = ASCII(2)
    ( MD5(), BIN(), HEX(), VERSION(), USER(), bit_length(), SPACE() ...)

1 OR 1 IS NOT NULL

1 OR NULL IS NULL

1' HAVING 1 #1 !

1' OR id=1 HAVING 1 #1 !

a'or-1='-1

a'or!1='!1

a'or!(1)='1

a'or@1='@1

a'or-1 XOR'0

1'OR!(false) #1 !

1'OR-(true) #a !

a' OR if(-1=-1,true,false)#!
```

```
1' OR 1&'1
1' OR 1|'1
1' OR 1^'1
1' OR 1%'1
1' OR '1' & 1
1' OR '1' && '1
1' OR '1' XOR '0
1' OR "1" ^ '0
1' OR '1' ^ '0
1' OR '1'|'2
1' OR '1' XOR '0

1 OR+1=1
1 OR+(1)=(1)
1 OR+'1'=(1)
1 OR+'1'=1
1 OR '1'!=0
```

What should a reliable WAF filter rule look like for this?

Exercise

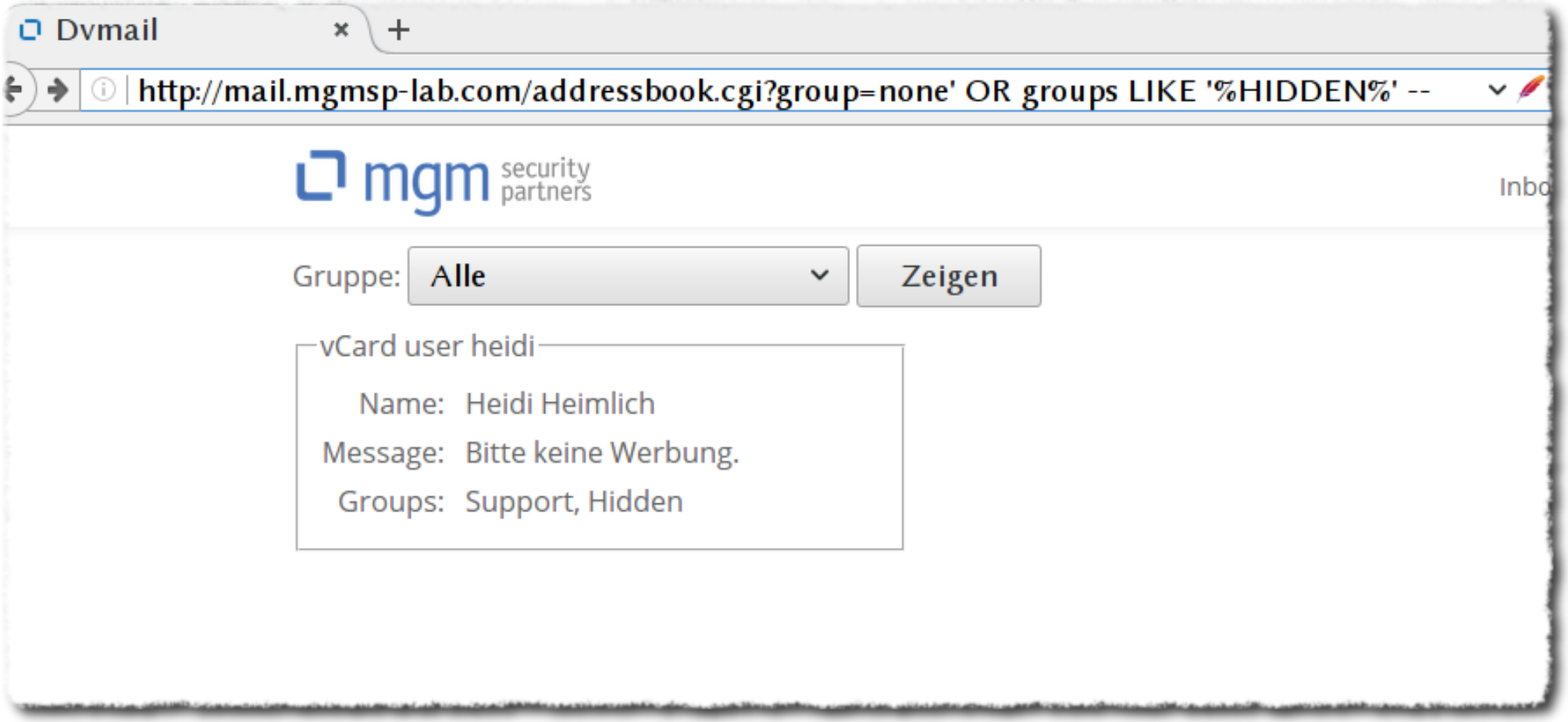
Exercise: SQL Injection

- Preparation
- Register an account at the **Mail**-App (linked from the Dashboard)

- Exercise
  1. Find the SQL Injection vulnerability in the address book
  2. Use the vulnerability to show all users (users within the group “hidden” are normally not displayed)

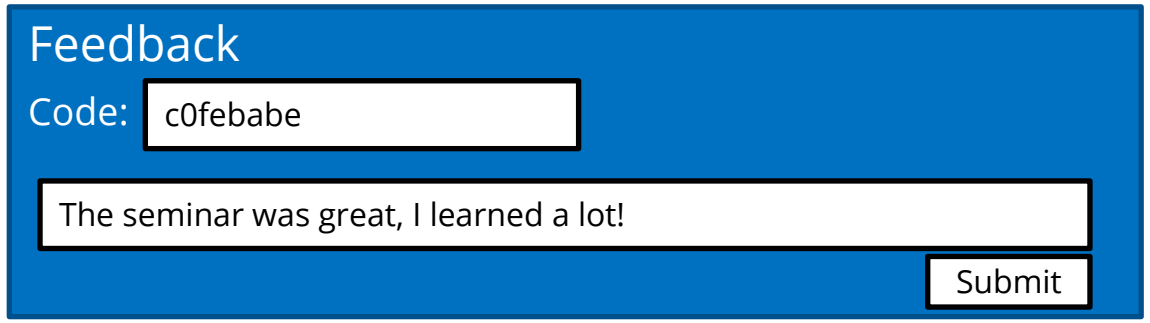
- Bonus Exercise
  - Use the SQL Injection to display all password hashes of all users
  - Can you find the password of Wolfgang S.?

Exercise: Solution

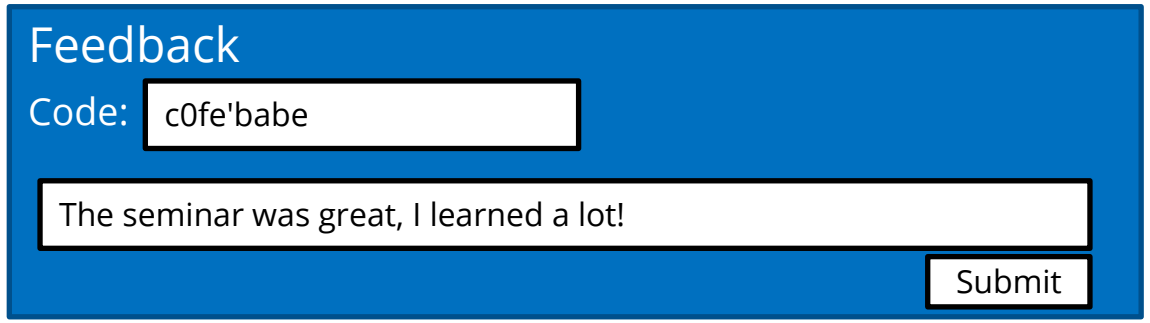


Blind SQL Injection

Blind SQL Injection



Thank you for your feedback!

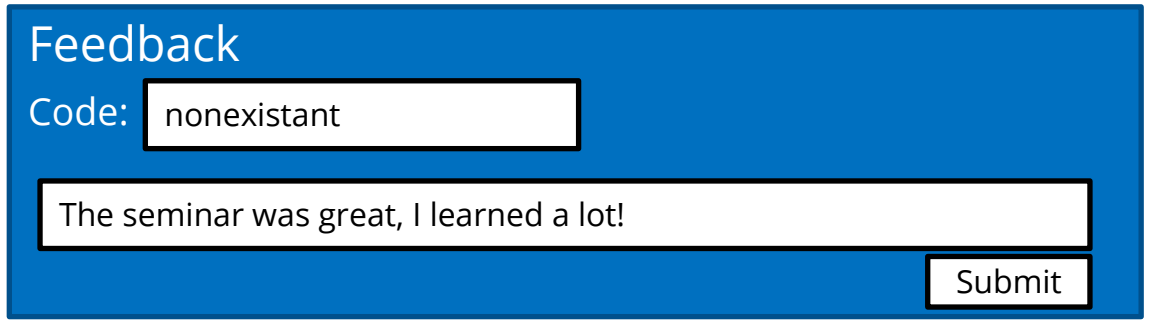


The feedback-code was not found.

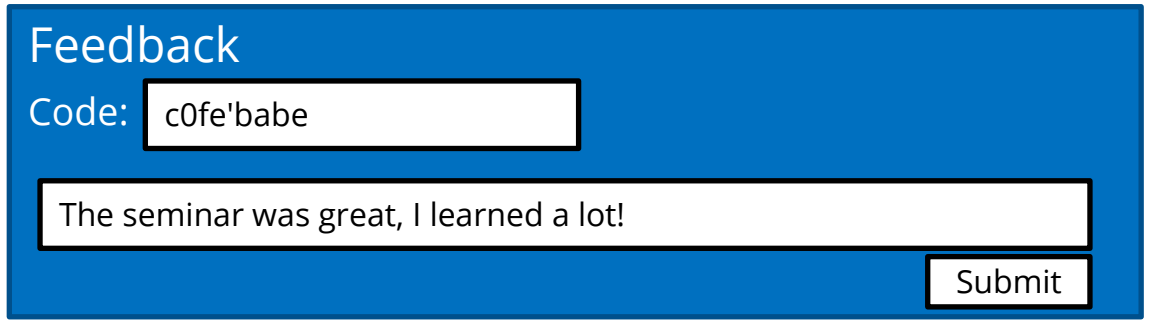
→ no error is thrown
possible (faulty) implementation:

```
try {
  s.executeQuery("UPDATE fb SET text='"+request.post("text")+"' WHERE id='"+request.post("code")+"'");
} catch (SQLException e) {
  // do nothing
}
```

Blind SQL Injection



The feedback-code was not found.

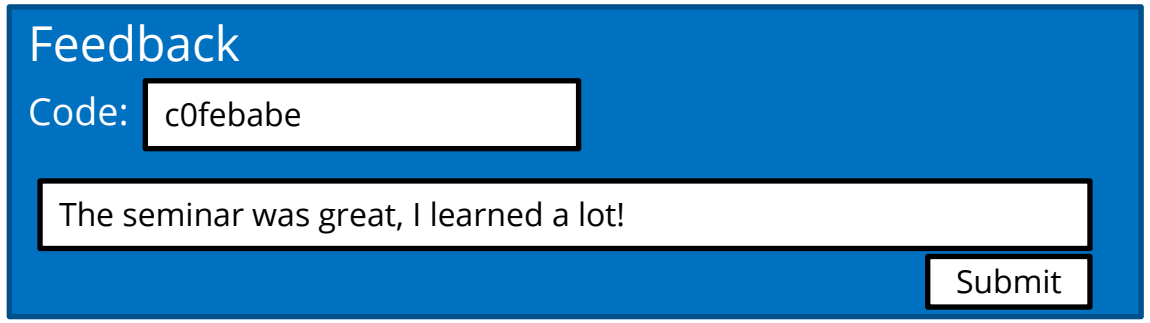


The feedback-code was not found.

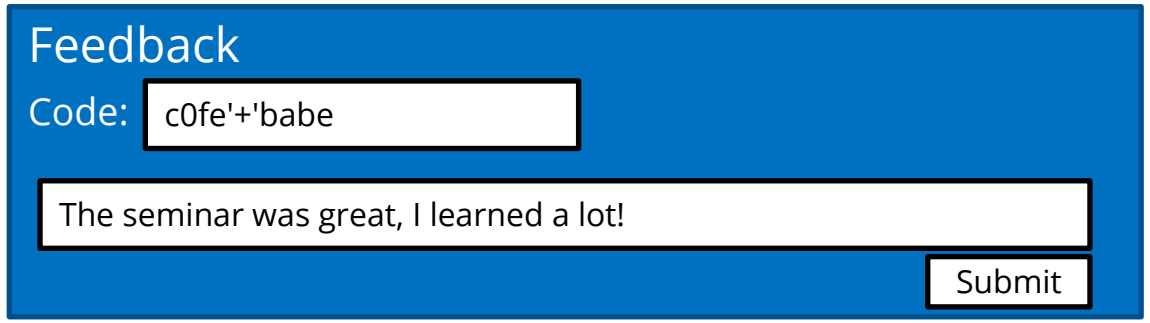
→ possible SQL Injection can not be distinguished from other false requests

Blind SQL Injection

Boolean based detection



Thank you for your feedback!



Thank you for your feedback!

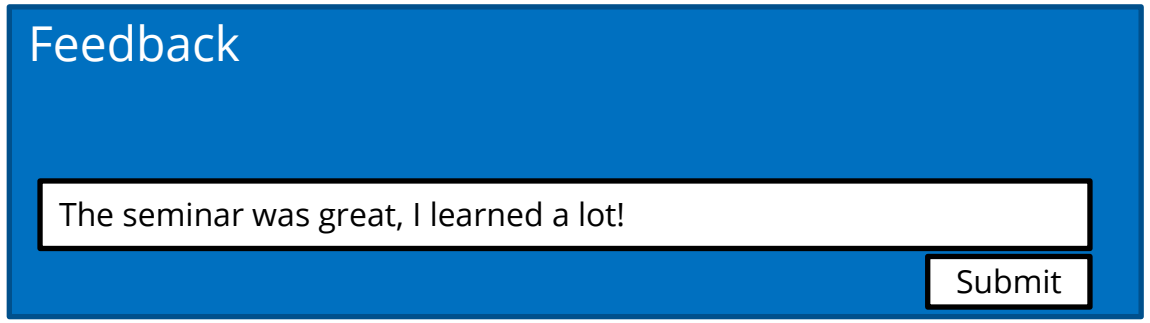
→ may be distinguished from requests which are rendered correctly!

```
UPDATE * fb SET text = 'The seminar was great, I learned a lot!' WHERE id = 'c0febabe'
UPDATE * fb SET text = 'The seminar was great, I learned a lot!' WHERE id = 'c0fe'+'babe'
```

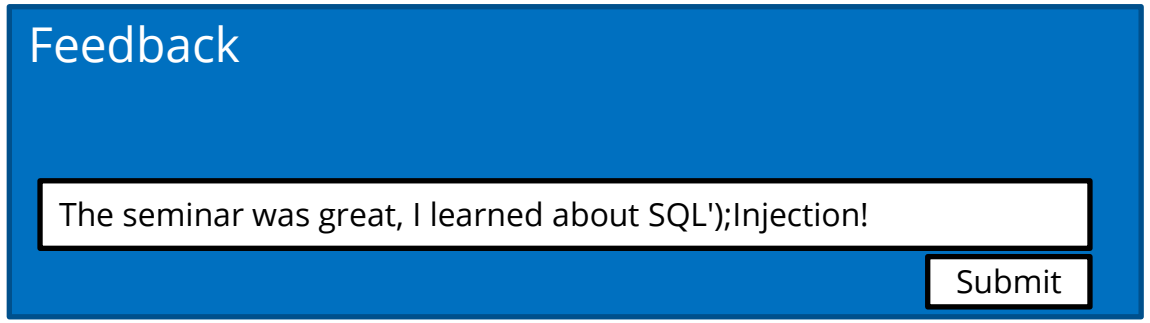
→ strong indication for SQL Injection

Blind SQL Injection

Full blind



Thank you for your feedback!



Thank you for your feedback!

→ maybe even fully blind without any output
possible (faulty) implementation:

```
try {
  s.executeQuery("INSERT INTO feedback (text) VALUES ('" + request.post("feedback") + "')");
} catch (SQLException e) {
  // do nothing
}
```

Blind SQL Injection

Extract data

- Idea:
  - use a side channel (aka out-of-band signalling)
  - E.g.: time-based side channel

```
SLEEP(IF(SELECT password FROM … )="Password123",5,0))
```

    - ➢ Delay of 5 seconds if password is guessed correctly

- Problem:
  - Slow
  - 1 decision per SELECT statement

Blind SQL Injection

Extract data

- On average 5,000 tries necessary

- If you could ask the lock the following questions:
  - **Does the code start with** 0 → no
  - Does the code start with 1 → no
  - Does the code start with 2 → yes
  - Does the code start with 20 → no
  - Does the code start with 21 → yes
  - …
  - Does the code start with 2140 → no
  - Does the code start with 2141 → yes

- … then one would just need 5×4 = 20 tries on average

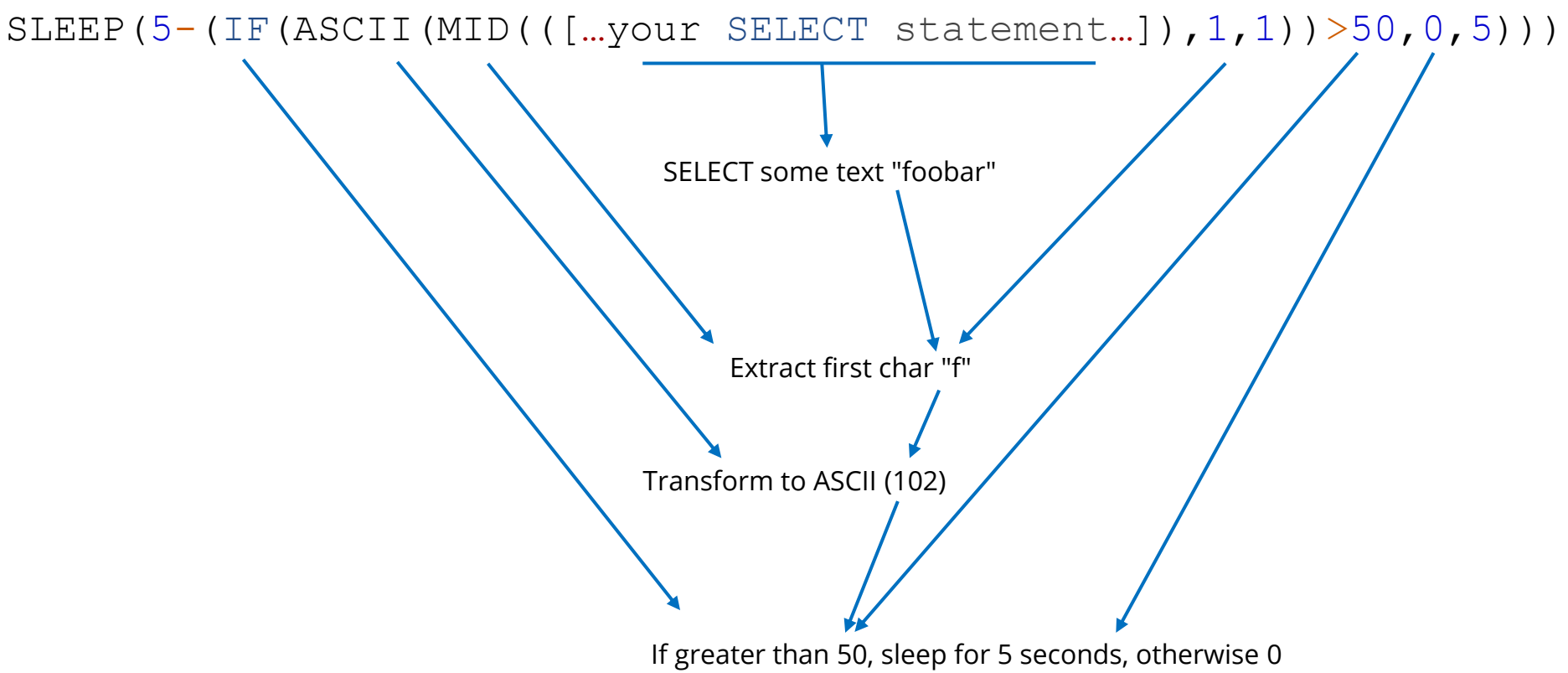
Blind SQL Injection – more efficient data extraction

Extract data, increase efficiency

- In fact, we can ask the lock the following questions:
  - Is the first digit less than 5 → yes
  - Is the first digit less than 3 → yes
  - Is the first digit less than 1 → no
  - Is the first digit 2 → yes
  - Is the second digit less than 5 → yes
  - …
  - Is the last digit less than 2 → yes
  - Is the last 1 → yes

- Binary instead of linear search → complexity of $O(\log_2(n))$

Blind SQL Injection – how to extract data



Exploiting OCR in license plate cameras …

SQL Injection

see also ...

- OWASP
  https://owasp.org/www-community/attacks/SQL_Injection

- SQL Injection Cheat Sheets
  http://pentestmonkey.net/category/cheat-sheet/sql-injection
  https://portswigger.net/web-security/sql-injection/cheat-sheet
  https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/

- sqlmap
  http://sqlmap.org/

SQL Injection Best Practices

SQL Injection

Countermeasures

- Prepared Statements

```
PreparedStatement p = connection.prepareStatement("SELECT id FROM data WHERE name = ?");
String custname = request.getParameter("name");
p.setString(1, custname);
```

- Stored Procedures

```
CREATE PROCEDURE GetId
    @username varchar(50)
AS
BEGIN
    SELECT id FROM data WHERE name = @username;
END
GO
```

- Object-relational Mappers

```
@Entity
public class User {
   @Id
   private int userId;
   private String userName;
...
```

```
public static void main(String[] args) {
   ...; EntityManager em = ...; ...;

   User u = new User(123,req.getParameter("name");
   em.persist(u);
...
```

- If dynamic SQL statements are required, DB-specific escaping needs to be applied, e.g.:

```
String name = "McHale's Navy";

// Oracle uses '' for escaping, % and _ are not escaped (used in LIKE only)
String escapedName = name.Replace("'", "''");
statement.executeQuery("SELECT id FROM data WHERE name = '" + escapedName + "'");
```

- Defense-in-Depth
  - Input Validation
  - Separated table spaces
  - Least privilege connections (database user having minimal access rights)

SQL Injection

Prepared Statements

Language - Library	Parameterized Query

Java - Standard	`String custname = request.getParameter("customerName");`
	`String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";`
	`PreparedStatement pstmt = connection.prepareStatement(query);`
	`pstmt.setString(1, custname);`
	`ResultSet results = pstmt.executeQuery();`

Java - Hibernate	`Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");` `safeHQLQuery.setParameter("productid", userSuppliedParameter);`

.NET/C#	`String query = "SELECT account_balance FROM user_data WHERE user_name = ?";`
	`try {  OleDbCommand command = new OleDbCommand(query, connection);`
	`command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));`
	`OleDbDataReader reader = command.ExecuteReader();`
	`// …`
	`} catch (OleDbException se) {`
	`// error handling`
	`}`

ASP.NET	`string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";`
	`SqlCommand command = new SqlCommand(sql);`
	`command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.SqlDbType.Int));`
	`command.Parameters["@CustomerId"].Value = 1;`

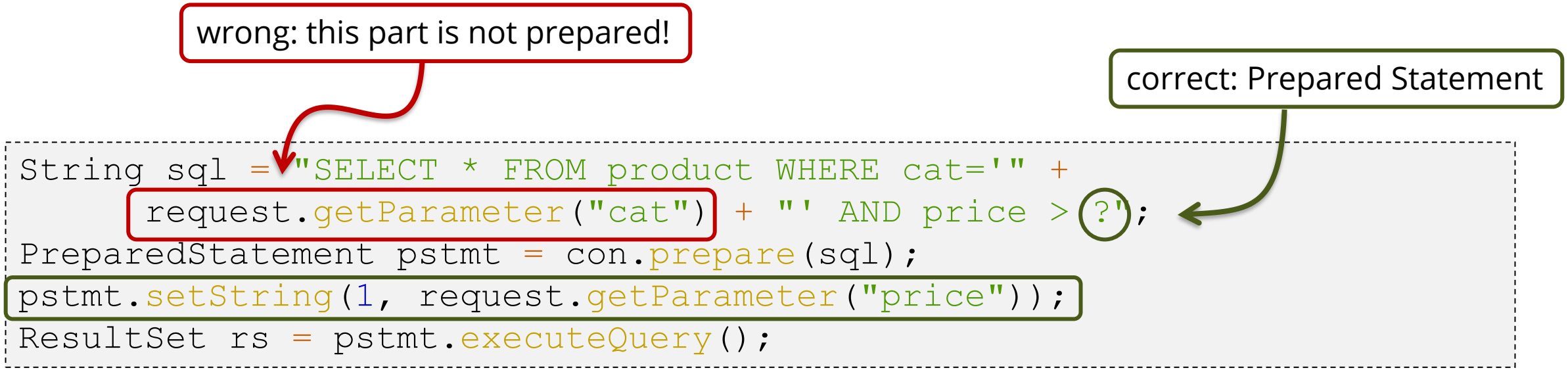
PHP - PDO	`$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");`
	`$stmt->bindParam(':name', $name);`
	`$stmt->bindParam(':value', $value);`

https://www.owasp.org/index.php/Abridged_SQL_Injection_Prevention_Cheat_Sheet

… in Prepared Statements, Stored Procedures

SQL Injection

...may also happen within Prepared Statements



SQL Injection

... is not impossible when using Stored Procedures

wrong

```
CREATE PROCEDURE VerifyUser
    @username varchar(50),
    @password varchar(50)
AS
BEGIN
    DECLARE @sql nvarchar(500);
    SET @sql = 'SELECT * FROM UserTable
                WHERE UserName = ''' + @username + '''
                AND Password = ''' + @password + ''' ';
    EXEC(@sql);
END
GO
```

Query is build dynamically and executed with EXEC...

good

```
CREATE PROCEDURE VerifyUser
    @username varchar(50),
    @password varchar(50)
AS
BEGIN

    SELECT * FROM UserTable
    WHERE UserName = @username
    AND Password = @password;

END
GO
```

... instead of writing it as a statement directly

Some more injection types...

OS Command Injection

OS Command Injection

2021 OWASP Top 10	
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failure
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery (SSRF)

2021 CWE Top 25 (MITRE)	
1	Out-of-bounds Write
2	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	Out-of-bounds Read
4	Improper Input Validation
5	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7	Use After Free
8	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	Cross-Site Request Forgery (CSRF)
10	Unrestricted Upload of File with Dangerous Type
11	Missing Authentication for Critical Function
12	Integer Overflow or Wraparound
13	Deserialization of Untrusted Data
14	Improper Authentication
15	NULL Pointer Dereference
16	Use of Hard-coded Credentials
17	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	Missing Authorization
19	Incorrect Default Permissions
20	Exposure of Sensitive Information to an Unauthorized Actor
21	Insufficiently Protected Credentials
22	Incorrect Permission Assignment for Critical Resource
23	Improper Restriction of XML External Entity Reference
24	Server-Side Request Forgery (SSRF)
25	Improper Neutralization of Special Elements used in a Command ('Command Injection')

mgm

OS Command Injection

Python

- Injection may be possible using several characters, e.g.:
  `;` `$` `` ` `` `&` ...

insecure

```
import os

os.system("/bin/echo insecure " + param)
```

- User input is encapsulated in an own parameter

secure

```
import subprocess

subprocess.run([
    "/bin/echo",
    "secure",
     param
])
```

- **Warning**: Specification of options may still be possible, e.g.: `param = "-e"`

NoSQL Injection

NoSQL Queries

- Queries are typically constructed using objects, not strings
- Examples (PHP+MongoDB):
  - SQL: `SELECT * FROM db WHERE foo = 'bar'`
  - NoSQL: `$db->find(['foo' => 'bar'])`

  - SQL: `SELECT * FROM db WHERE id != 3`
  - NoSQL: `$db->find(['id' => ['$ne' => 3]])`

  - SQL: `SELECT * FROM db WHERE foo = 'bar' OR spam = 'ham'`
  - NoSQL: `$db->find(['$or' => [['foo' => 'bar'],['spam' => 'ham']]])`

- Where clause may be used with JavaScript function
  - `$db->find(['$where' => "function() { return foo == 'bar'; }"]);`

NoSQL Injection

JavaScript Injection

- JavaScript-Code-Injection may be done in the **`$where`** clause

```
$db->find(['$where' => "function() { return foo == '$_POST[bar]'; }"])
```

```
x'||'a'=='a
```

```
function() { return foo == 'x'||'a'=='a'; }
```

Webservices / XML

Webservice Vulnerabilities

XML Injection

- Batch job defined via XML using user input:

```
<batchjob>
    <payment>
        <account>5678-attacker</account>
        <rcpt>206-1234</rcpt>
        <amount>100.00</amount>
        <comment>Placeholder for user input </comment>
    </payment>
</batchjob>
```

```
                    </comment>
    </payment>
    <payment>
        <account>1234-victim</account>
        <rcpt>206-1234</rcpt>
        <amount>100.00</amount>
        <comment>
```

https://www.owasp.org/images/5/5d/XML_Exteral_Entity_Attack.pdf

Webservice Vulnerabilities

XML Injection

- Following job is being transferred to the backend:

```
<batchjob>
    <payment>
        <account>5678-attacker</account>

        <rcpt>206-1234</rcpt>

        <amount>100.00</amount>

        <comment></comment>
    </payment>
    <payment>
        <account>1234-victim</account>

        <rcpt>206-1234</rcpt>

        <amount>100.00</amount>

        <comment></comment>
    </payment>
</batchjob>
```

https://www.owasp.org/images/5/5d/XML_Exteral_Entity_Attack.pdf

XPath

Injection-Attacks

XPath-Injection

- Erroneous access to a XML-database

Example file

```xml
<?xml version="1.0" encoding="utf-8"?>
<Employees>
    <Employee ID="1">
        <FirstName>Arnold</FirstName>
        <LastName>Baker</LastName>
        <UserName>ABaker</UserName>
        <Password>SoSecret</Password>
        <Type>Admin</Type>
    </Employee>
    <Employee ID="2">
        <FirstName>Peter</FirstName>
        <LastName>Pan</LastName>
        <UserName>PPan</UserName>
        <Password>NotTelling</Password>
        <Type>User</Type>
    </Employee>
</Employees>
```

Access per XPath using provided username and password:

C#

```
Expr = "//Employee[UserName/text()='" + Request("Username") + "' And Password/text()='" + Request("Password") + "']";
```

Input of username=ABaker and password=SoSecret

```
//Employee[UserName/text()='ABaker' And Password/text()='SoSecret']
```

https://owasp.org/www-community/attacks/XPATH_Injection

Injection-Attacks

XPath-Injection

- Erroneous access to a XML-database

Example file

```xml
<?xml version="1.0" encoding="utf-8"?>
<Employees>
    <Employee ID="1">
        <FirstName>Arnold</FirstName>
        <LastName>Baker</LastName>
        <UserName>ABaker</UserName>
        <Password>SoSecret</Password>
        <Type>Admin</Type>
    </Employee>
    <Employee ID="2">
        <FirstName>Peter</FirstName>
        <LastName>Pan</LastName>
        <UserName>PPan</UserName>
        <Password>NotTelling</Password>
        <Type>User</Type>
    </Employee>
</Employees>
```

Access per XPath using provided username and password:

C#

```
Expr = "//Employee[UserName/text()='" + Request("Username") + "' And Password/text()='" + Request("Password") + "']";
```

Input of username=`foobarbaz' or 1=1 or 'a'='a` and password=`somedefinitelywrongpassword`

```
//Employee[UserName/text()='foobarbaz' or 1=1 or 'a'='a' And Password/text()='somedefinitelywrongpassword']
```

That is logically equivalent to:

```
//Employee[(UserName/text()='foobarbaz' or 1=1)or('a'='a' And Password/text()='somedefinitelywrongpassword')]
```

https://owasp.org/www-community/attacks/XPATH_Injection

Injection-Attacks

XPath-Injection

Solution: use compiled expressions

Example file

```xml
<?xml version="1.0" encoding="utf-8"?>
<Employees>
    <Employee ID="1">
        <FirstName>Arnold</FirstName>
        <LastName>Baker</LastName>
        <UserName>ABaker</UserName>
        <Password>SoSecret</Password>
        <Type>Admin</Type>
    </Employee>
    <Employee ID="2">
        <FirstName>Peter</FirstName>
        <LastName>Pan</LastName>
        <UserName>PPan</UserName>
        <Password>NotTelling</Password>
        <Type>User</Type>
    </Employee>
</Employees>
```

Access per XPath using provided username and password:

C#

insecure

```
Expr = "//Employee[UserName/text()='" + Request("Username") + "' And Password/text()='" + Request("Password") + "']";
```

secure

```
XPathExpression Expr = "//Employee[UserName/text()=$user And Password/text()=$pass]";

DynamicContext Dc = new DynamicContext();
Dc.AddVariable("user", Request("Username"));
Dc.AddVariable("pass", Request("Password"));
Expr.SetContext(Dc);
```

https://learn.microsoft.com/en-us/dotnet/standard/data/xml/compiled-xpath-expressions

Header Poisoning / Header Injection

Header Poisoning

- Header Poisoning
  - Manipulating the **Request Header**
    ➔ Applications sometimes assume that the (honest) browser sets the headers.
    - **`Referer/Host/X-Forwarded-For`** used for access control
    - **`Referer/User-Agent`** is logged and displayed as HTML (➔ XSS!)
    - **`Cookie`** is considered as „set by the server → clean“
    - Buffer Overflow by overlong headers
  - Manipulation of the **Response Headers**
    ➔ Goal: targeted attack on the client
  - Many headers are potentially exploitable – depending on how the respective application uses them.

Header Injection

The Location-Response-Header

- https://www.example.org/redirect?url=/new-page.html

Browser Request:

```
GET /redirect?url=/new-page.html HTTP/1.1
Host: www.example.org
```

Server Response:

```
HTTP/1.1 302 Found
Location: /new-page.html
```

CRLF

Header Injection

The Location-Response-Header

- https://www.example.org/redirect?url=/new-page.html%0d%0aX-Set:%20111

Browser Request:

```
GET /redirect?url=/new-page.html%0d%0aX-Set=111 HTTP/1.1
Host: www.example.org
```

Server Response:

```
HTTP/1.1 302 Found
Location: /new-page.html
X-Set: 111
```

%0d%0a

CRLF

Header Injection

The Location-Response-Header

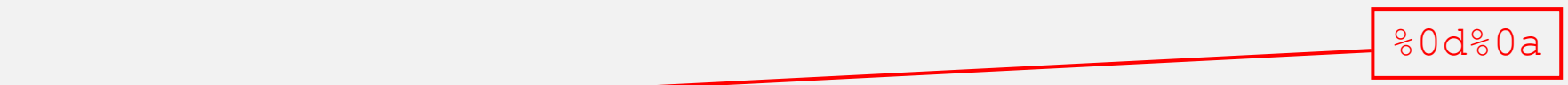
- https://www.example.org/redirect?url=/new-page.html%0d%0aSet-Cookie:%20JSESSIONID=0000QU3TX4T0HGUBXVHM22I0DZA%3b%20expires=Friday,%2031-Dec-2099%2023:59:59 GMT%3b%20domain=.example.org

Browser Request:

```
GET /redirect?url=/new-page.html%0d%0aSet-Cookie:%20JSESSIONID=0000QU3TX4T0HGUBXVHM22I0DZA%3b%20expires=Friday,%2031-Dec-2012 23:59:59%20GMT%3b%20domain=.example.org HTTP/1.1
...
```

Server Response:

```
HTTP/1.1 302 Found
Location: /new-page.html
Set-Cookie: JSESSIONID=0000QU3TX4T0HGUBXVHM22I0DZA; expires=Friday, 31-Dec-2099 23:59:59 GMT; domain=.example.org
```



Header Injection

- Always an issue when user-data is being placed into header
- Redirect-Vulnerability:
  - Does not occur for new server software
  - Hand-made redirects can be susceptible!
  - Setting of cookies (Set-Cookie directive) / Content-Security-Policy Bypass
- Effects
  - Faking the SessionID (Session Fixation, Session-DoS)
  - HTTP Response Splitting / HTTP Request Smuggling
  - Manipulation of loadbalancers and other infrastructure components

- %0d%0a or \015\012 must be filtered out of the input string

Injection General

Injection

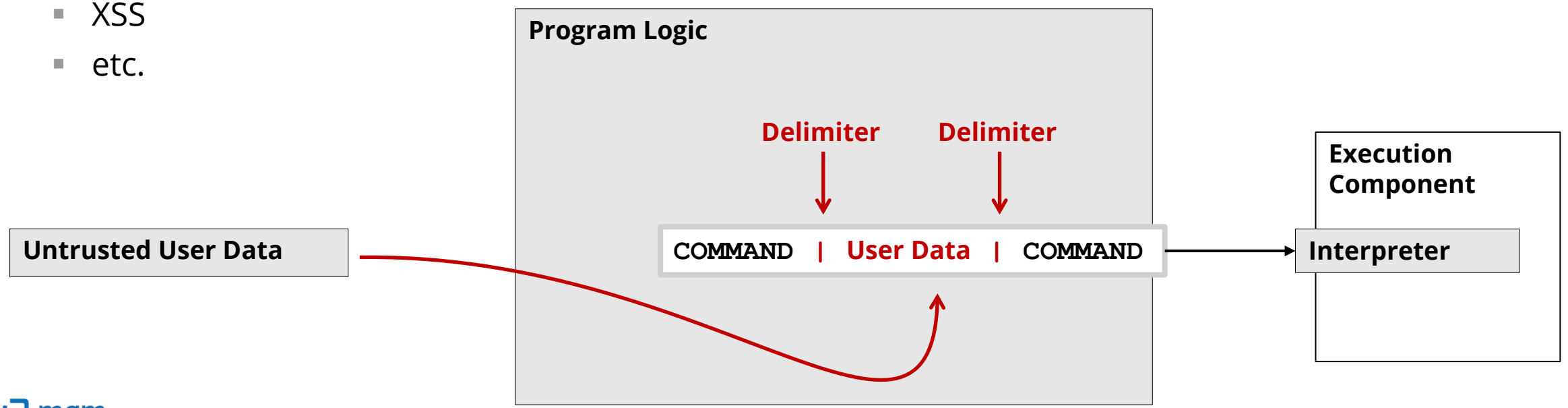
2021 OWASP Top 10	
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Component
7	Identification and Authentication Failu
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failu
10	Server-Side Request Forgery (SSRF)

2021 CWE Top 25 (MITRE)	
1	Out-of-bounds Write
2	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	Out-of-bounds Read
4	Improper Input Validation
5	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7	Use After Free
8	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	Cross-Site Request Forgery (CSRF)
10	Unrestricted Upload of File with Dangerous Type
11	Missing Authentication for Critical Function
12	Integer Overflow or Wraparound
13	Deserialization of Untrusted Data
14	Improper Authentication
15	NULL Pointer Dereference
16	Use of Hard-coded Credentials
17	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	Missing Authorization
19	Incorrect Default Permissions
20	Exposure of Sensitive Information to an Unauthorized Actor
21	Insufficiently Protected Credentials
22	Incorrect Permission Assignment for Critical Resource
23	Improper Restriction of XML External Entity Reference
24	Server-Side Request Forgery (SSRF)
25	Improper Neutralization of Special Elements used in a Command ('Command Injection')

mgm

The Injection Pattern

- The injection problem occurs in many places
  - SQL Injection
  - XML Injection
  - XPath Injection
  - LDAP Injection
  - CMD Injection
  - XSS
  - etc.

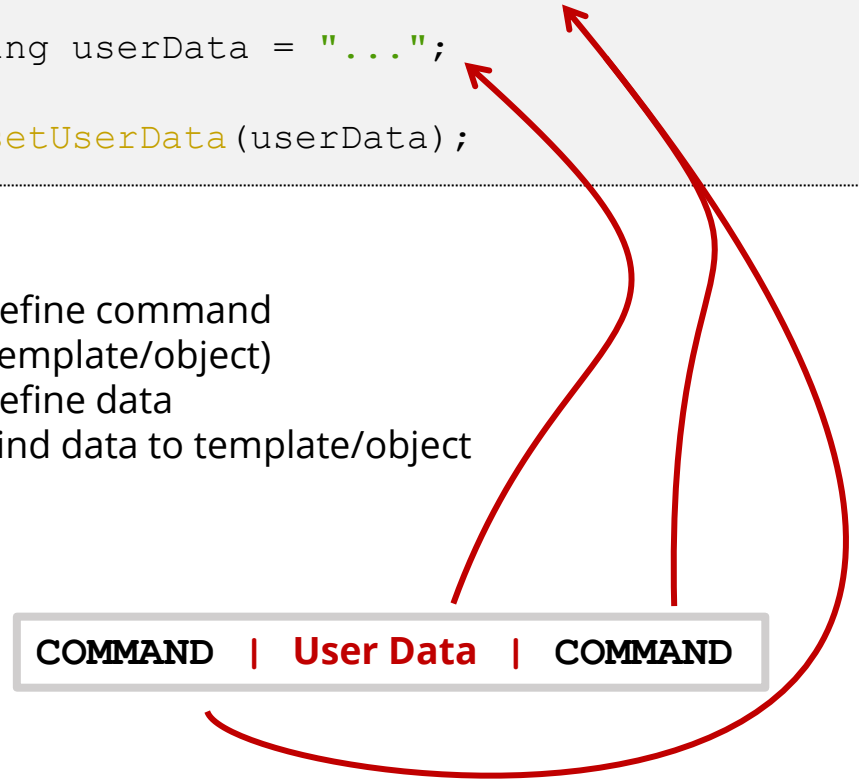


Common Prevention

Separate Code and Data

```
String command  = "...";
op = Something.constructOperator(command);

String userData = "...";

op.setUserData(userData);
```

1. Define command (template/object)
2. Define data
3. Bind data to template/object



wrong

```
String custname = request.getParameter("name");
String query = "SELECT id FROM data WHERE name = '"+custname+"'";
```

good

```
String query = "SELECT id FROM data WHERE name = ?";
PreparedStatement pstmt = connection.prepareStatement(query);

String custname = request.getParameter("name");


pstmt.setString(1, custname);
```

wrong

```
exec("mail -f file.eml -t -s '" + getSubject() + "'")
```

good

```
mailSubject = getSubject()

system("mail", "-f", "file.eml", "-t", "-s", mailSubject)
```

wrong

```
elem.innerHTML = "<input type='text' value='"+fetchUserData()+"'>"
```

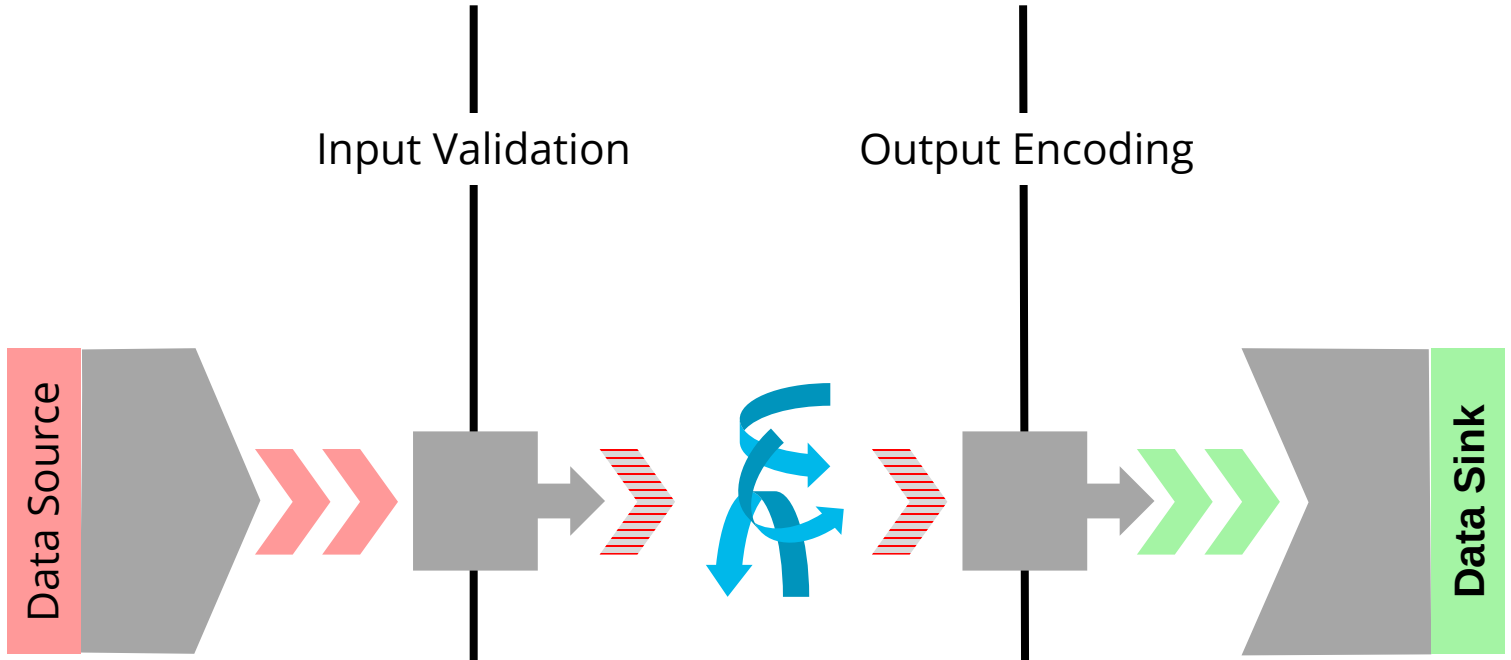
good

```
var x = document.createElement("input");

var userData = fetchUserData();

x.setAttribute("value", userData);
```

mgm

Data Validation

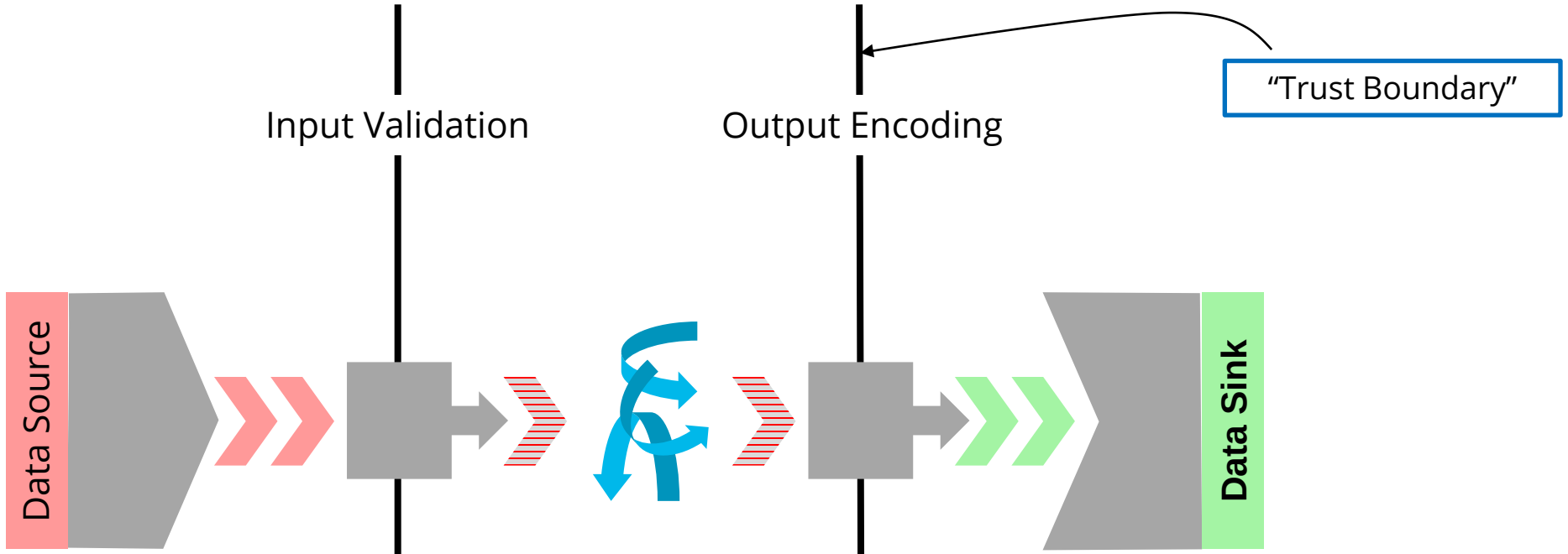
Input and Output Handling



e. g.

Browser (User)		Data Base
Data Base		File
File		Process
Process		Browser
Trust Level:		
untrusted	**untrusted or trusted**	**trusted**

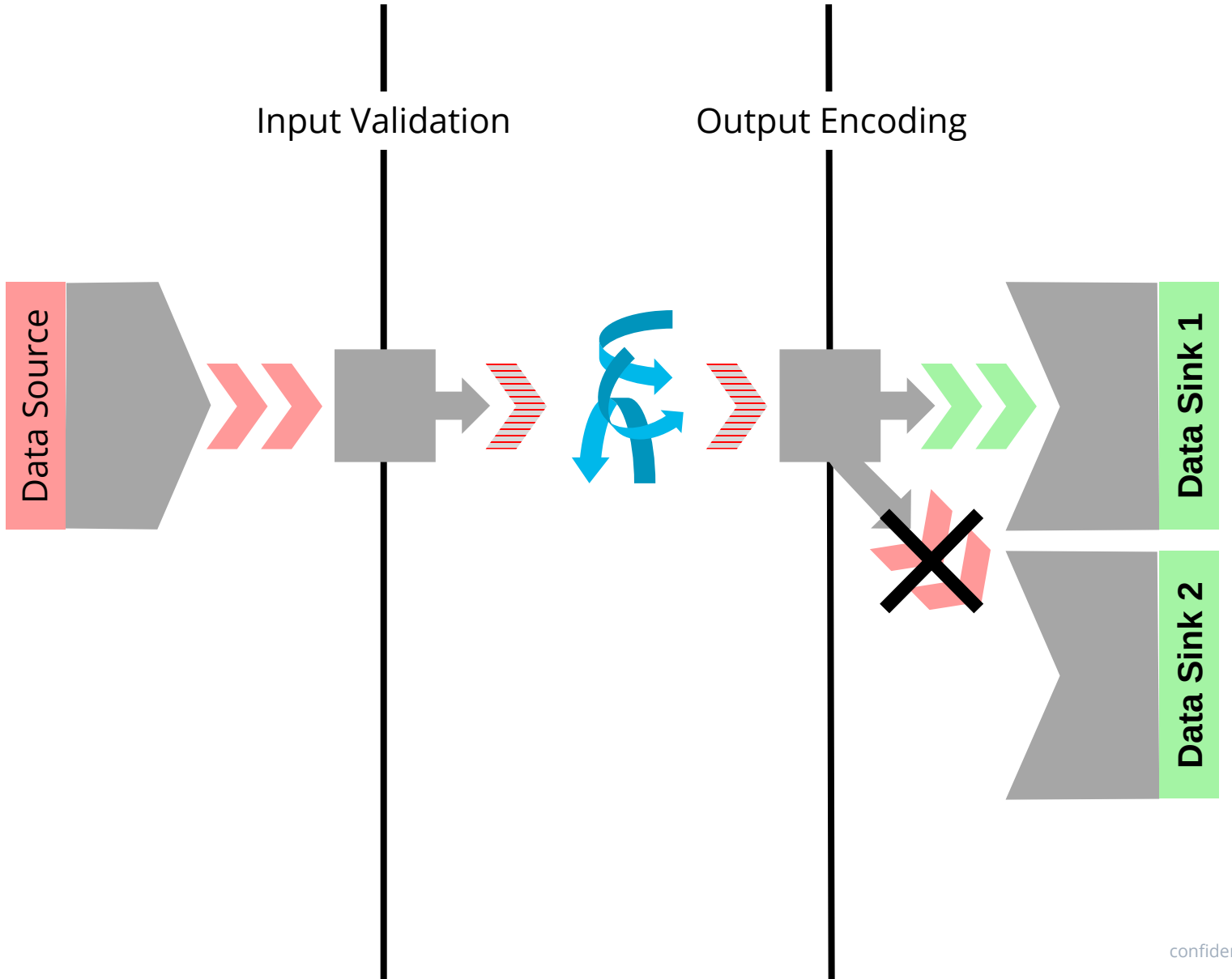
Input and Output Handling



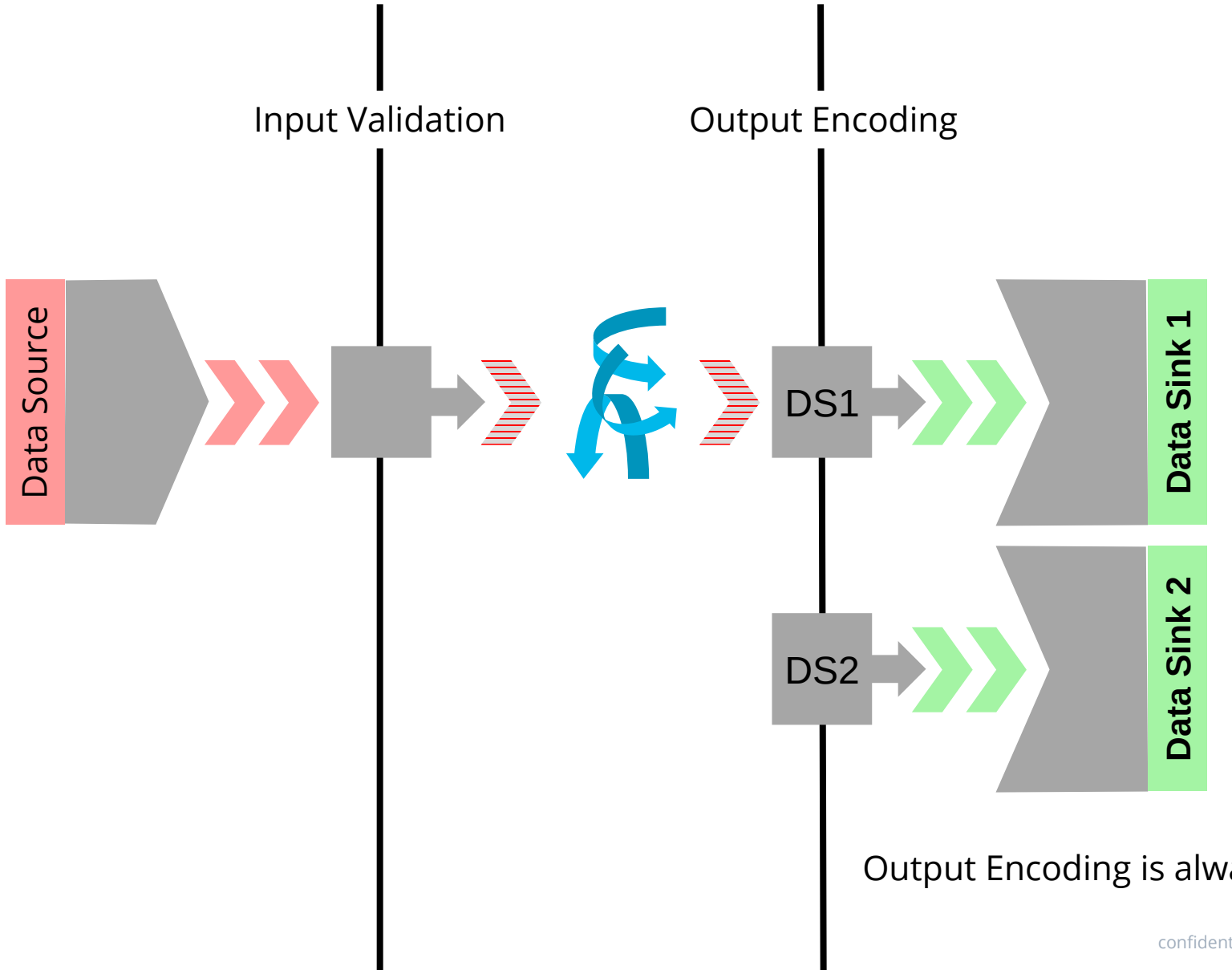
e. g.

Browser (User)		Data Base
Data Base		File
File		Process
Process		Browser
Trust Level:		
untrusted	**untrusted or trusted**	**trusted**

Input and Output Handling

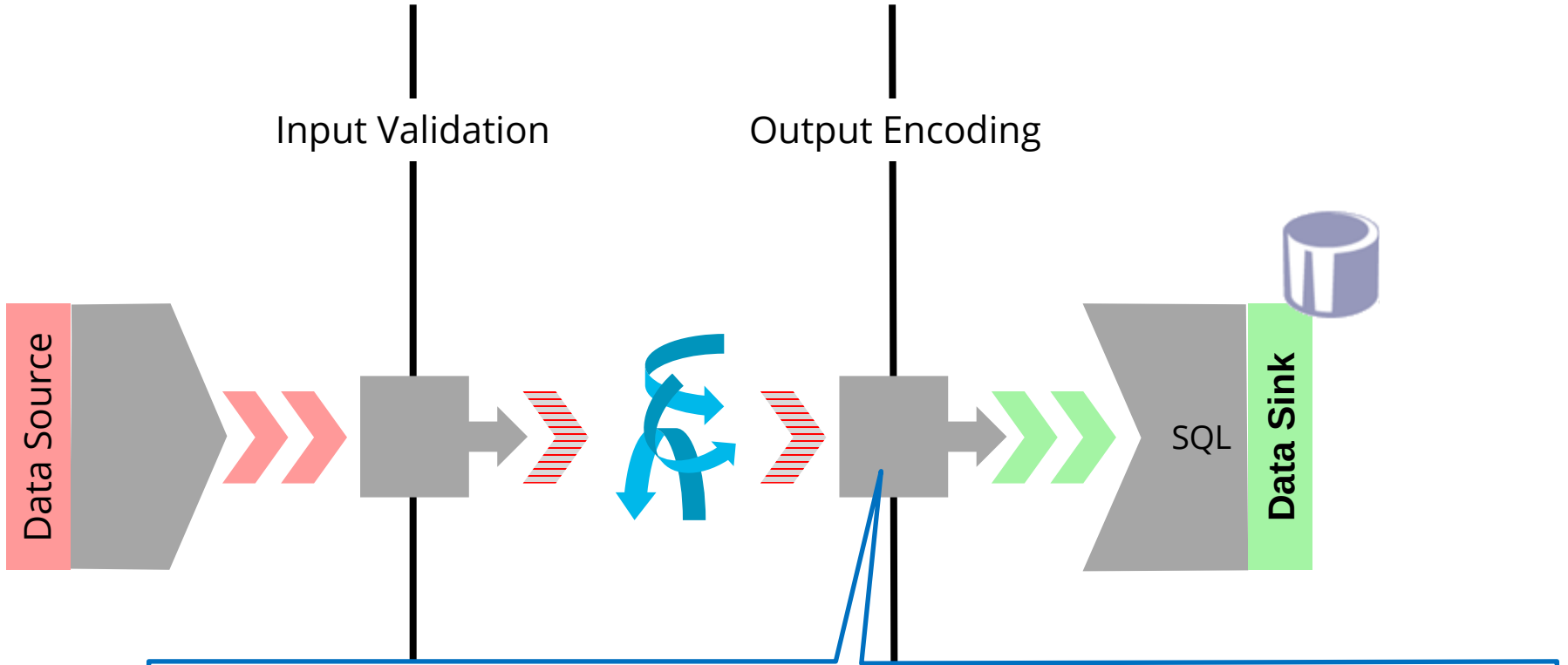


Input and Output Handling



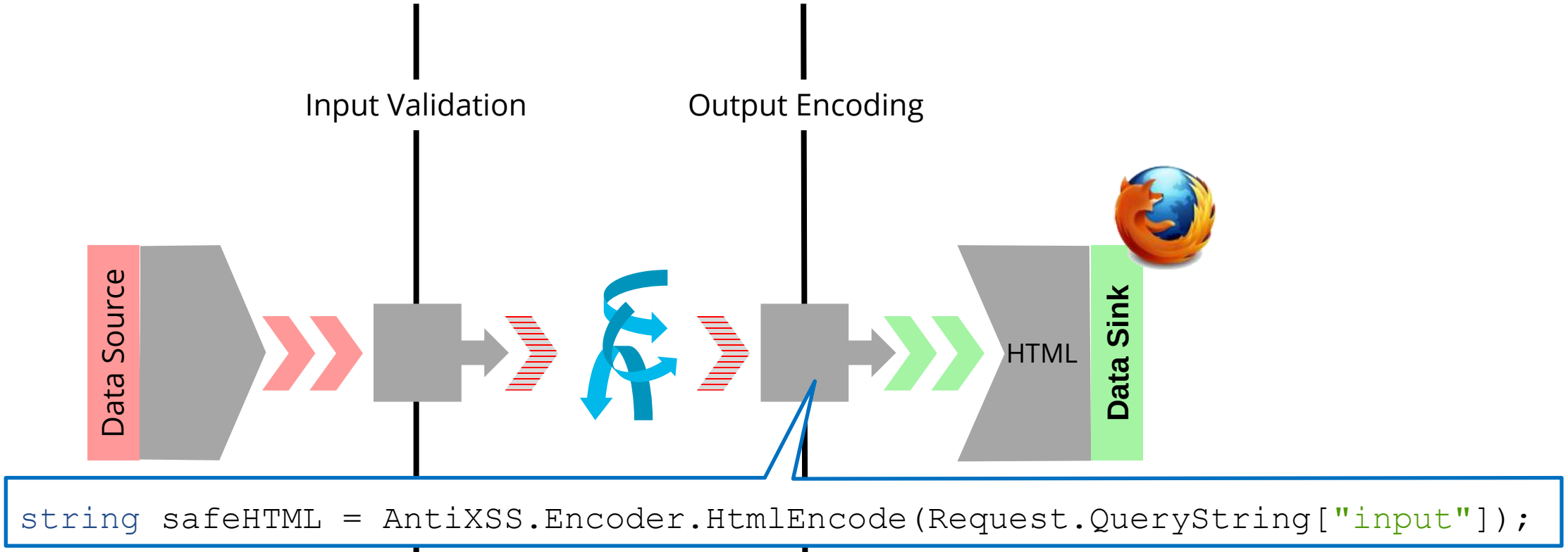
Output Encoding is always context-driven!

Input and Output Handling

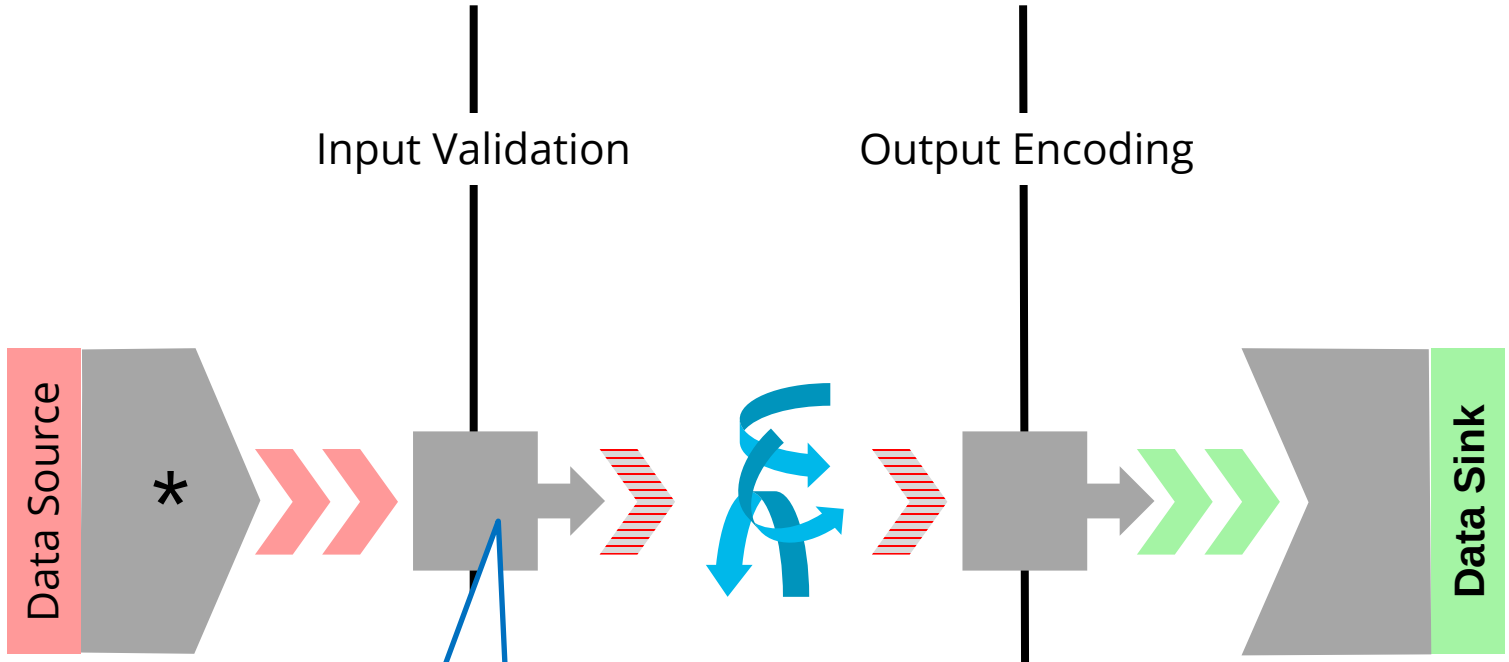



```
stmt.executeQuery("SELECT * FROM t WHERE TITLE='" +
    StringEscapeUtils.escapeSql("O'Connor") + "'");
```

Input and Output Handling



Input and Output Handling



e.g.

```
id   = getInteger(input);     // Integer
cc   = getCreditCard(input); // [0-9]{14}
zip  = getZipCode(input);    // e.g. D-12345
```

but also:

```
email   = getEmail(input);   // valid'OR'1'='1'--@rfc-5322.mail
comment = getText(input);    // String + maxlength + sanitize unwanted chars/tags…
```

Summary

Summary

Injection

- “All input is evil until proven otherwise”
- Separate code and data
- Avoid dynamic Statements
- Input type checking + output encoding
- Prefer whitelists

- Further measures
  - Simplicity
  - Defensive Programming
  - Least privilege
  - Defense in depth
  - Layer of indirection

Quiz

SQL Injection

What is the main idea of an SQL-Injection attack?

A. It is possible to submit single ticks (') from the outside and therefore one may use special characters.
B. Because the ? in prepared statements can interfere from the outside with special characters, it is possible to trick the interpreter.
C. Because the interpreter will only get a string which includes user input and developer code, it can not distinguish both parts.
D. The eval statement of stored procedures is not safe for user input. Therefore an escaping has to be done outside the procedure-statement.

SQL Injection

Which character would be problematic, if the following code is implemented:

```
query = "SELECT user FROM employees WHERE id = '" + id + "' OR group = 'admin'"
```

A. single tick: '
B. quotation mark: "
C. parenthesis: ( )
D. Space
E. comment: "-- "

SQL Injection

Review the following file:

https://github.com/mgm-sp/NinjaDVA-quiz/blob/15bb9052ab74d7c5cd00ce6ca3aad89923ffb22e/api/src/main/java/quiz/Api.java

At which line can you identify an SQL-Injection?

A. 63
B. 99
C. 105
D. 114
E. 151
F. 163

SQL Injection

Why is it a bad practice to show error messages and stack traces in production?

A. Attackers will get internal information about the code
B. Developers will be able to debug their code more efficient
C. In case of an error, end-users are able to give more information in a support case.
D. A+B+C

SQL Injection

Why is it a good practice to give db users minimal rights?

A. Because it is needed for compliancy reasons
B. If there is a misconfiguration, it will do less damage
C. An attacker who finds an SQL Injection will be able to read out only parts of the database
D. A+B+C

SQL Injection

Why is it a good practice to give the OS-user, which is used to execute the database, minimal permissions?

A. To ensure system integrity if the database runs out of memory
B. To ensure, that logfiles are written to the correct place
C. An attacker who is able to find an SQL Injection should be limited in his actions

Injection

What is the best way in order to prevent Injection attacks in general?

A. Validate all Input fields to their specification in a least-privilege principle
B. Encode all Output with the respective encoding
C. Do the query in a way, where you separate the query-code and the userdata