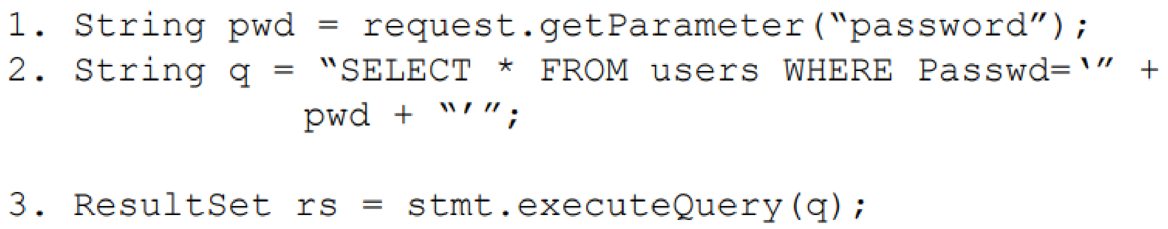
C8 Web-based Code Injections I

1. SQL Injection

1.1 Overview

SQL Injection

- Server-side script (may) construct SQL query as a string from query fragments and user inputs before passing to DBMS to get executed as a SQL query
    - DMBS is able to interpret user input as code due to broken abstraction
- Example SQL Query



```
String pwd = request.getParameter(“password”);
String q = “SELECT * FROM users WHERE Passwd=‘” +
              pwd + “’”;

ResultSet rs = stmt.executeQuery(q);
```

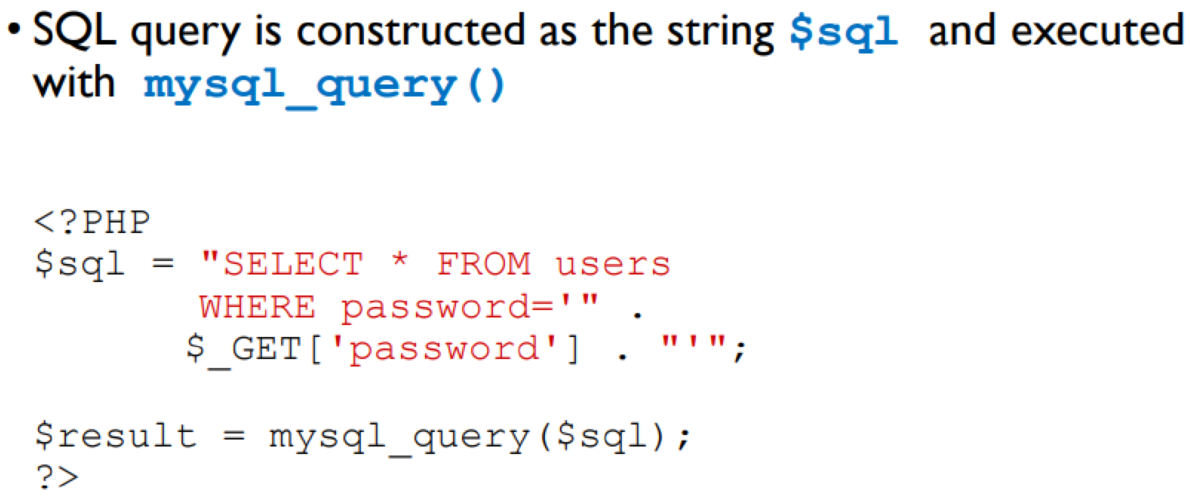
    - `pwd` comes from user input → user can supply malicious input as `pwd` variable (e.g. `' OR 1=1`)
    - Attacker is able to return entire Users table
    - Attacker can also supply additional commands in the input (e.g. `xyz'; drop table Users`) which leads to deletion of the Users table

1.2 Defences

1. Getting rid of single quotes (poor)

- Done by escaping with a backslash (`\'`)
- PHP check functions
    - addslashes() - applied by default
    - mysqli_real_escape_string()

    - is_numeric()

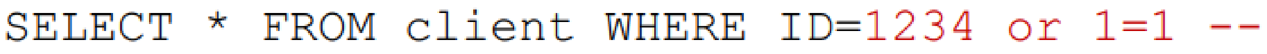


- SQL query is constructed as the string `$sql` and executed with `mysql_query()`

```
<?PHP
$sql = "SELECT * FROM users
       WHERE password='" .
      $_GET['password'] . "'";

$result = mysql_query($sql);
?>
```

- Sanitizing user inputs
- **Comment operator (`--`)** comments out any characters added to the query by the code



```
SELECT * FROM client WHERE ID=1234 or 1=1 --
```

2. Looking for predefined attack patterns (poor)

- `1=1` attacks are sought after by intrusion detection systems and firewalls
- Using any other tautologies would work just as effectively

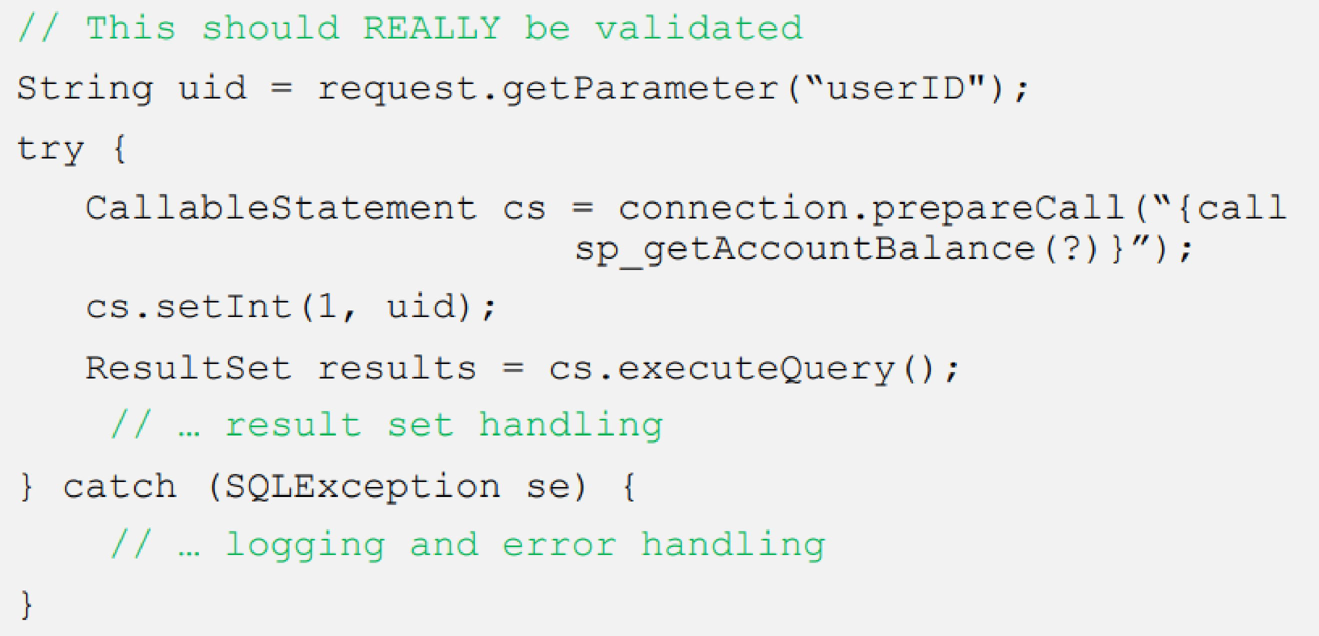
3. Using prepared statements

- Tell DBMS which part of the SQL query is data and which part is code
    - Script is compiled with placeholders instead of user input
    - Replace placeholders by actual user input when executing compiled script

4. Using stored procedures securely

- Stored procedures are defined and stored in the DB itself and called from the application

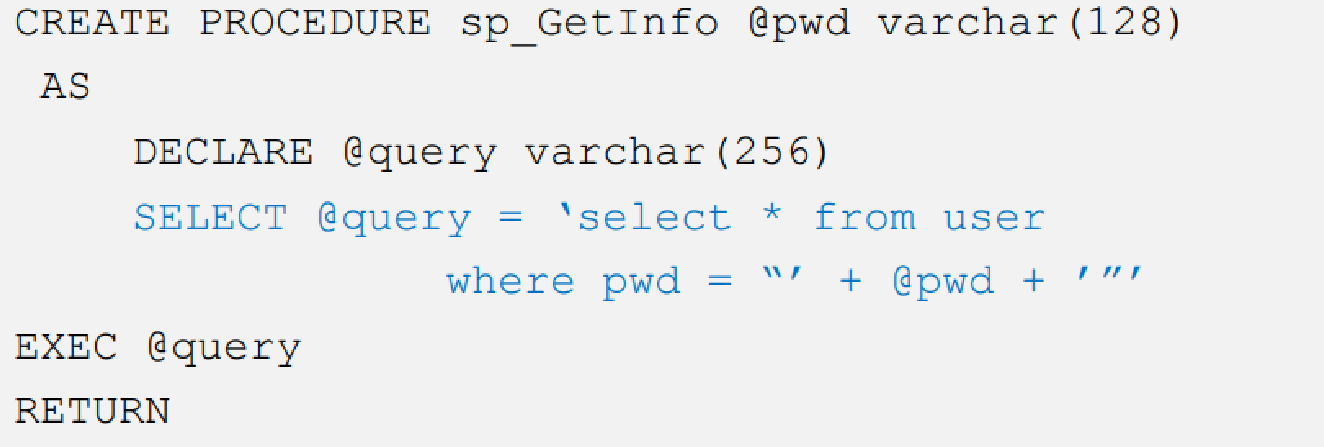
- Same effectiveness in preventing SQL injections when implemented **securely**



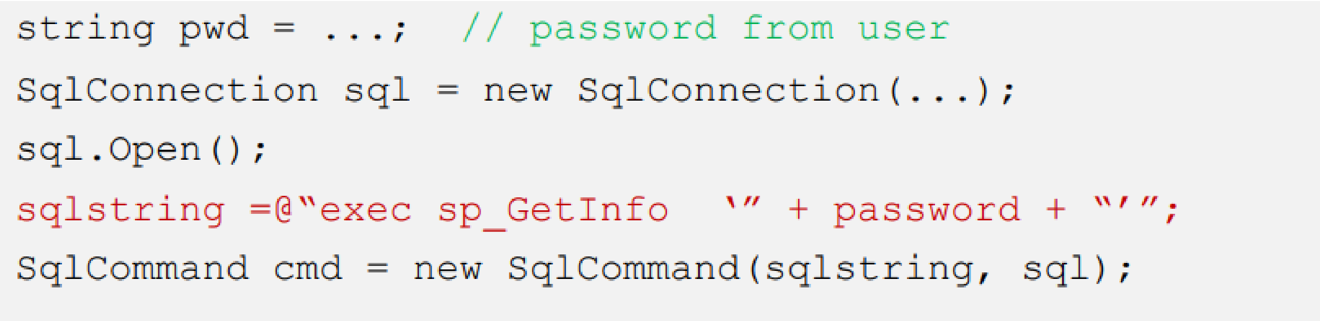
```
// This should REALLY be validated
String uid = request.getParameter(“userID");
try {
   CallableStatement cs = connection.prepareCall(“{call
                             sp_getAccountBalance(?)}”);
   cs.setInt(1, uid);
   ResultSet results = cs.executeQuery();
    // … result set handling
} catch (SQLException se) {
    // … logging and error handling
}
```

5. Stored procedures with dynamic queries (poor)

- Insecure implementation of stored procedures



```
CREATE PROCEDURE sp_GetInfo @pwd varchar(128)
 AS
     DECLARE @query varchar(256)
     SELECT @query = ‘select * from user
                  where pwd = “’ + @pwd + ’”’
EXEC @query
RETURN
```



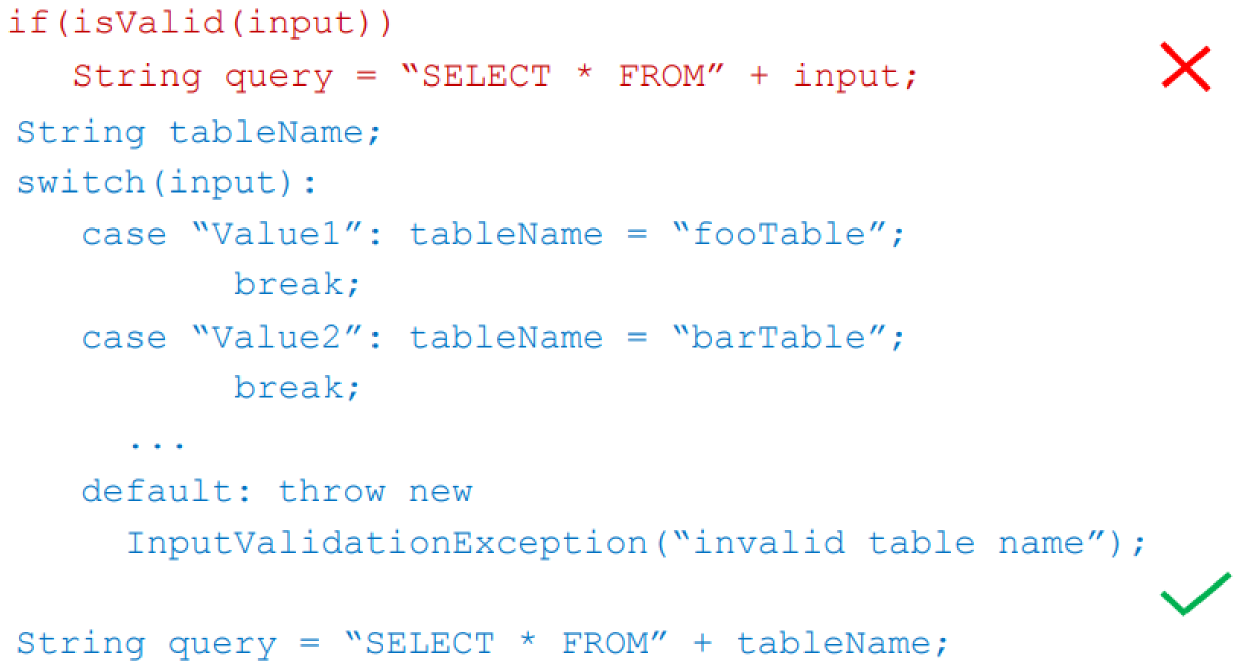
```
string pwd = ...;  // password from user
SqlConnection sql = new SqlConnection(...);
sql.Open();
sqlstring =@“exec sp_GetInfo  ‘” + password + “’”;
SqlCommand cmd = new SqlCommand(sqlstring, sql);
```

(`password` should be `pwd`)

  - `xyz' or 1=1 --` no longer works because `or 1=1--` should not be called right after a stored procedure since it is not a complete boolean expression
  - `xyz' insert into client values (1005, 'Mike')` would however work
    - A second SQL query can be added to the end of the stored procedure

6. Whitelist input validation

- Bind variables are not useful for some parts of SQL queries such as names of tables or columns; input validation / query redesign becomes the most appropriate defence
  - Names of tables and columns should come from code and not user inputs
- **Table name validation**

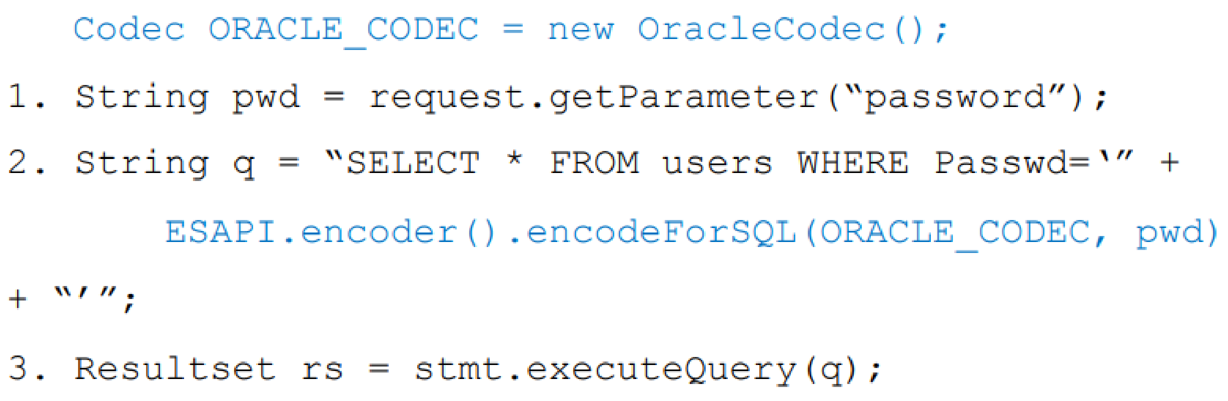


```
if(isValid(input))
   String query = “SELECT * FROM” + input;              ✗
String tableName;
switch(input):
   case “Value1”: tableName = “fooTable”;
          break;
   case “Value2”: tableName = “barTable”;
          break;
     ...
   default: throw new
     InputValidationException(“invalid table name”);
                                                          ✓
String query = “SELECT * FROM” + tableName;
```

  - Table is chosen from input value and not user provided (e.g. user is only given options and each option is mapped to a table)
- Input validation using RegEx

7. Escaping

- Cannot guarantee to prevent all SQL Injections
  - e.g. encoding attacks / inputs are used in `LIKE` clauses
- Use when parameterised commands are not feasible / to complement input validation



```
   Codec ORACLE_CODEC = new OracleCodec();
String pwd = request.getParameter(“password”);
String q = “SELECT * FROM users WHERE Passwd=‘” +
       ESAPI.encoder().encodeForSQL(ORACLE_CODEC, pwd)
+ “’”;
Resultset rs = stmt.executeQuery(q);
```

- `pwd` is replaced with `ESAPI.encoder().encodeForSQL(ORACLE_CODEC, pwd)`
- **Escaping WildCard characters**
  - WildCards are used in conjunction with the `LIKE` operator
    - `%` - represents 0, 1 or multiple characters

        - `_` - represents a single character

2. Writing Secure Code

2.1 Defence in Depth

- Apply least privilege principle
- Encrypt stored data in case of breach
- Do not expose too much information in case of errors or exceptions

2.2 Least Privilege Principle

- Do not connect database as sysadmin from applications (most web-based applications do not need capabilities of sysadmin to run)
- **Access Control**
    - Determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away
        - Accounts that only need read access are only granted read access
        - If only a portion of table is required, creating a view would be better
        - Application accounts should be restricted to only executing stored procedures they need
- **Avoid verbose error messages**
    - Error messages for invalid inputs can leak information about database, relations, names of columns, etc.
    - Use proper error / exception handling code

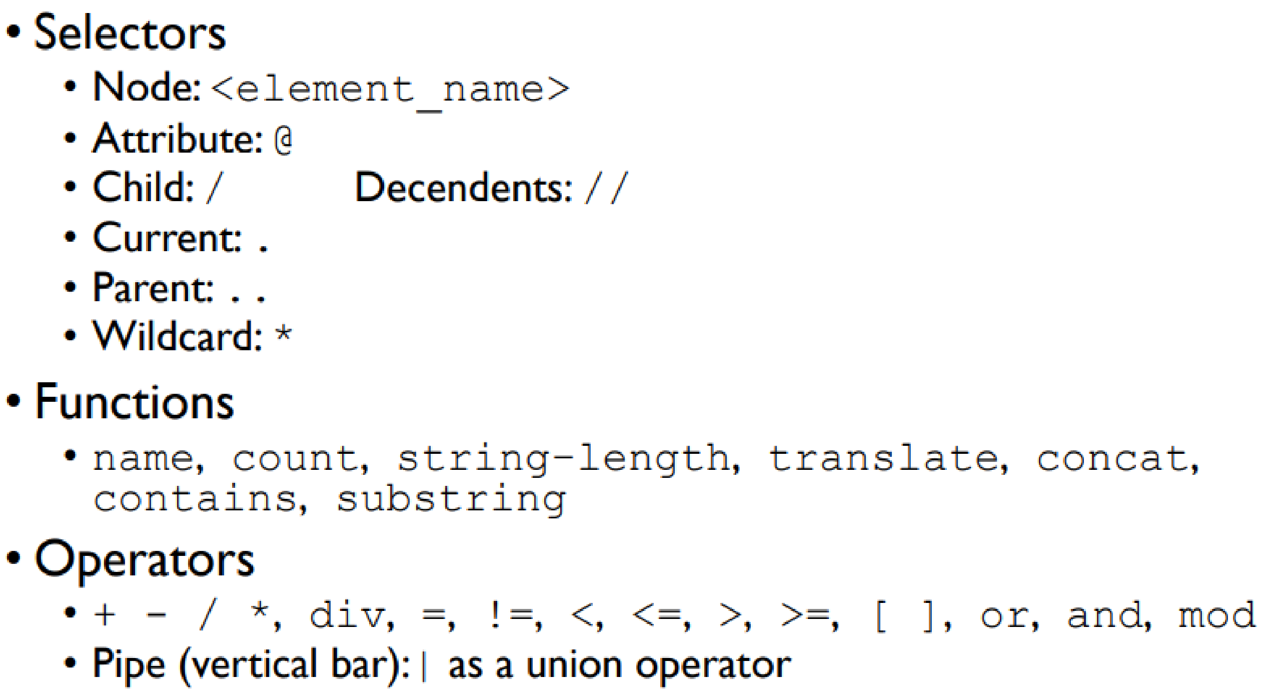
      ```
      try {
          //stmts that may cause an exception
      } catch (exception(type) e(object)) {
          //error handling code
      }
      ```

3. Other code injections

3.1 XML Path Language (XPath)

- Used to navigate through elements and attributes in XML document using a "path like" syntax (e.g. `/books/book/title`)

Terminology



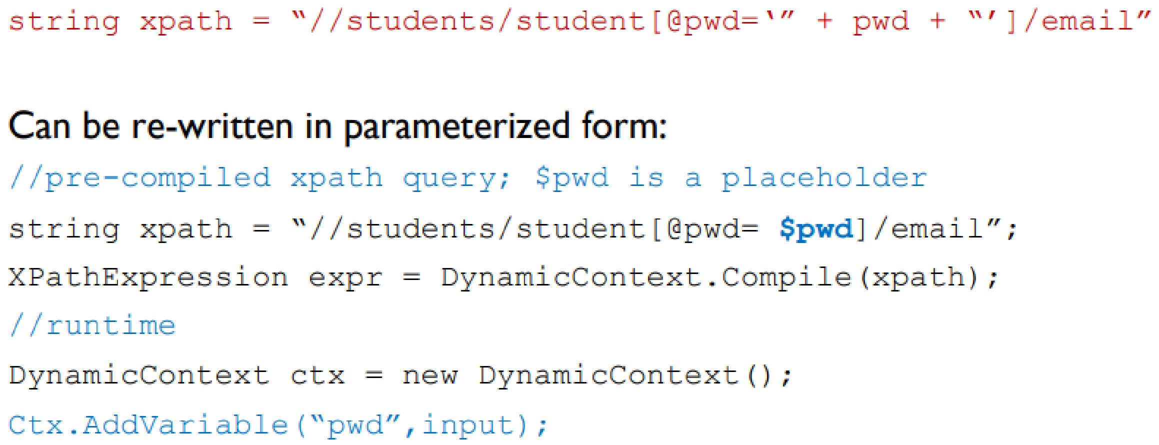
- Selectors
  - Node: `<element_name>`
  - Attribute: `@`
  - Child: `/` Decendents: `//`
  - Current: `.`
  - Parent: `. .`
  - Wildcard: `*`
- Functions
  - `name, count, string-length, translate, concat, contains, substring`
- Operators
  - `+ - / *, div, =, !=, <, <=, >, >=, [ ], or, and, mod`
  - Pipe (vertical bar): `|` as a union operator

Injection

- Attack can be conducted by placing meta characters into input string which alters the behaviour of the original query by modifying the query logic or bypassing authentication
- Low attack complexity since there are no different dialects like in SQL
- No level access control, possible to get entire document

Defenses

- Same as techniques to avoid SQL injections, use parameterised XPath interface if available (best option)



```
string xpath = “//students/student[@pwd='” + pwd + “']/email”
```

Can be re-written in parameterized form:

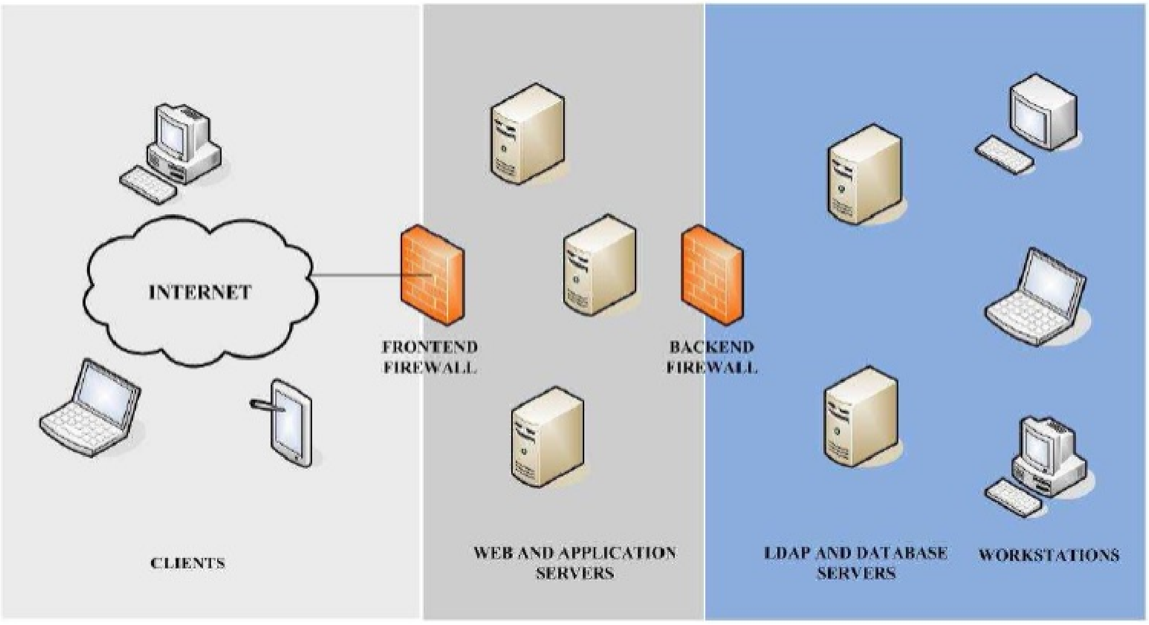
```
//pre-compiled xpath query; $pwd is a placeholder
string xpath = “//students/student[@pwd= $pwd]/email”;
XPathExpression expr = DynamicContext.Compile(xpath);
//runtime
DynamicContext ctx = new DynamicContext();
Ctx.AddVariable(“pwd”,input);
```

- Next best option is to validate and escape (OWASP ESAPI library)

3.2 Lightweight Directory Access Protocol (LDAP)

- Protocol for querying and modifying directory services
- Stores information about users, systems, networks, services, applications, etc.

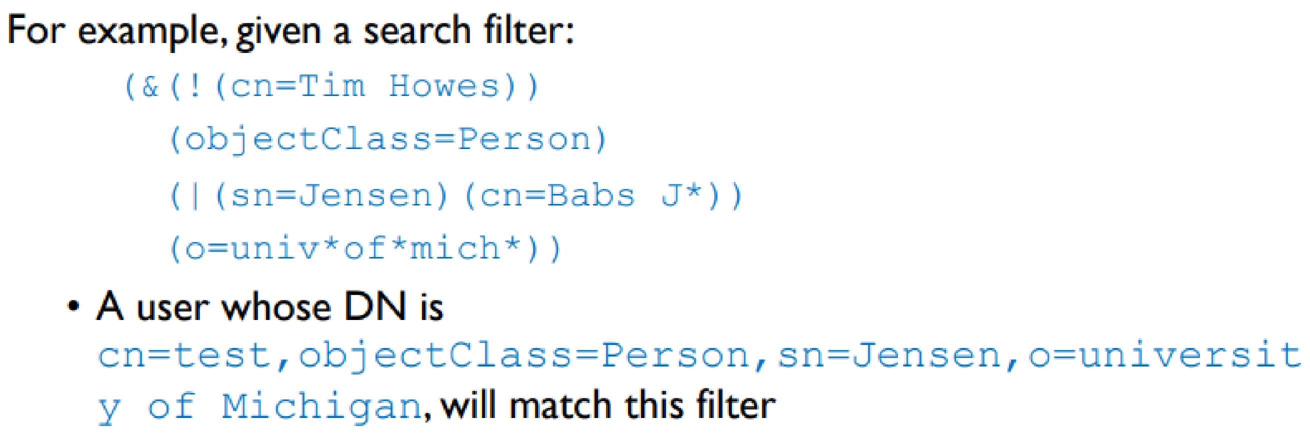
- Provides a central place for authentication and authorisation for users



- LDAP uses DNs (distinguished names) as entries in the directory; each DN points to exactly one entry (row)

Search filter

- Functions like a SQL `SELECT` query, uses operators
  - logical: (`&, |, !`)
  - relational: (`=, >=, <=, etc`)
  - special character: `*` that matches one or more characters
- Special constants `&` = absolute True, `|` = absolute False



For example, given a search filter:

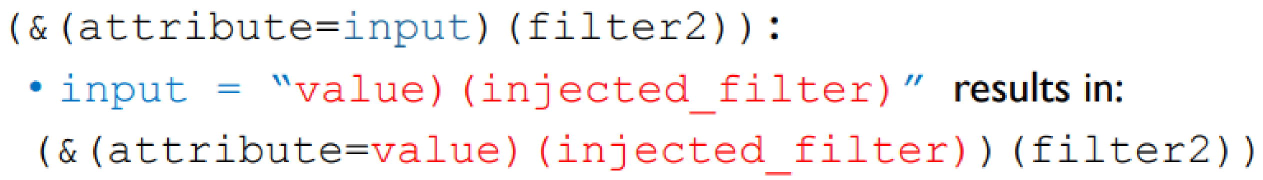
```
    (&(!(cn=Tim Howes))
      (objectClass=Person)
      (|(sn=Jensen)(cn=Babs J*))
      (o=univ*of*mich*))
```

- A user whose DN is `cn=test,objectClass=Person,sn=Jensen,o=university of Michigan`, will match this filter

Injection

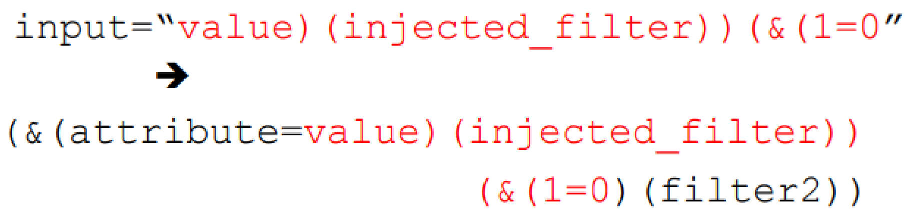
- Done by manipulating filters used to search in the directory services by inserting meta-characters into inputs passed into internal search, add, and modify functions which alter the logic of the query
- Permissions end up being granted to unauthorised queries or for modifying LDAP trees

- **Search filter injection:**



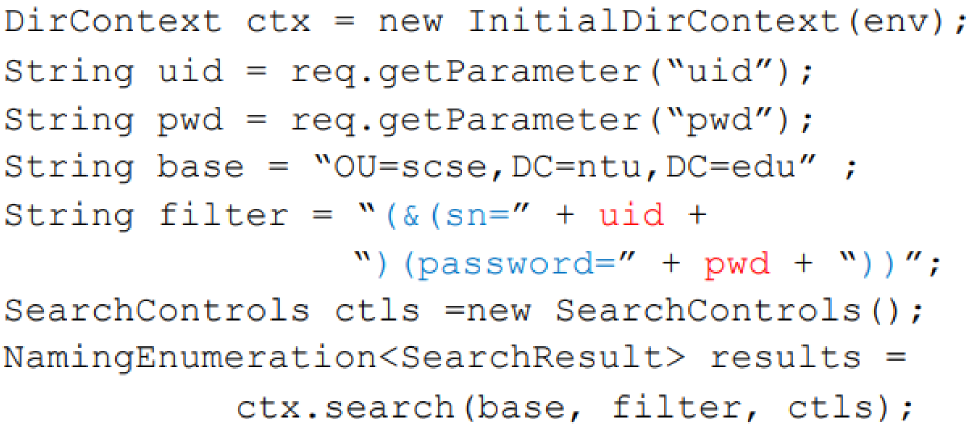
```
(&(attribute=input)(filter2)):
  • input = “value)(injected_filter)” results in:
 (&(attribute=value)(injected_filter))(filter2))
```

  - OpenLDAP and some LDAP web clients will ignore `filter2` even though syntactically incorrect
  - If syntactical correctness is checked, an adjustment can be made to `input`

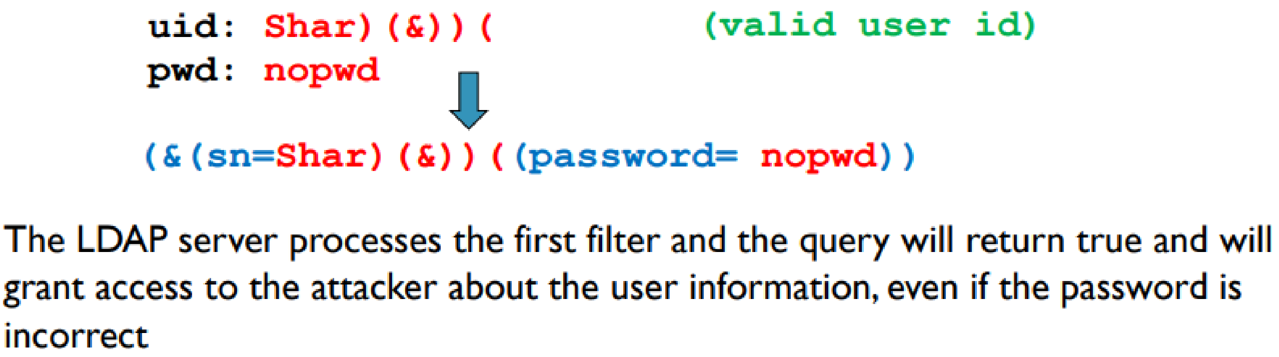


```
input=“value)(injected_filter))(&(1=0”
        ➔
(&(attribute=value)(injected_filter))
                                (&(1=0)(filter2))
```

- **Bypassing authentication**



```
DirContext ctx = new InitialDirContext(env);
String uid = req.getParameter(“uid”);
String pwd = req.getParameter(“pwd”);
String base = “OU=scse,DC=ntu,DC=edu” ;
String filter = “(&(sn=” + uid +
                  “)(password=” + pwd + “))”;
SearchControls ctls =new SearchControls();
NamingEnumeration<SearchResult> results =
          ctx.search(base, filter, ctls);
```



```
uid:  Shar)(&))(          (valid user id)
pwd:  nopwd
              ⇩
(&(sn=Shar)(&))((password= nopwd))
```

The LDAP server processes the first filter and the query will return true and will grant access to the attacker about the user information, even if the password is incorrect

- **Injection of wildcard**
  - For a search filter `(attribute = input)`, if `input` is `*`, filter becomes `(attribute = *)` which matches every value of the attribute

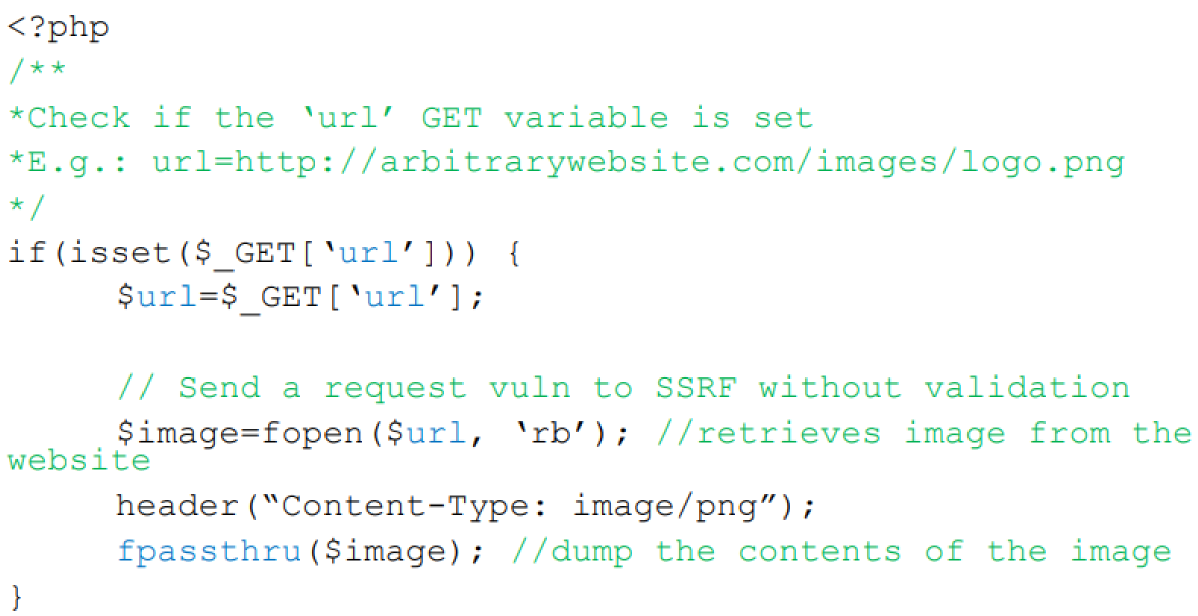
Defense

- Escape all variables using the right escaping function
  - Similar to SQLi but taking different set of meta-characters into account
  - LINQtoAD framework used for automatic escape

- Defense in depth, by whitelist input validation / not making LDAP server directly accessible on the Internet

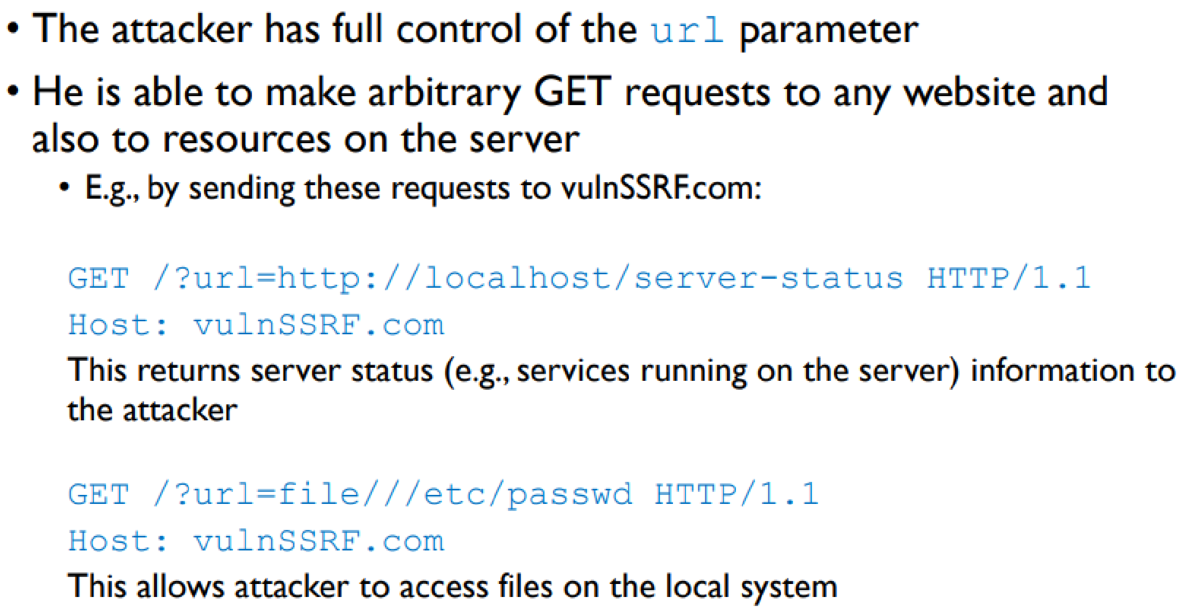
4. Server-side request forgery

- Occurs when a web application is making a request to internal systems, where an attacker has full / partial control of the request that is being sent
- Can be used for
  - Scanning other machines within the private network of vulnerable server that aren't externally accessible
  - Performing Remote File Inclusion attacks
  - Bypassing firewalls and using the vulnerable server to carry out malicious attacks
  - Retrieving server files



```
<?php
/**
*Check if the ‘url’ GET variable is set
*E.g.: url=http://arbitrarywebsite.com/images/logo.png
*/
if(isset($_GET[‘url’])) {
      $url=$_GET[‘url’];

      // Send a request vuln to SSRF without validation
      $image=fopen($url, ‘rb’); //retrieves image from the
website
      header(“Content-Type: image/png”);
      fpassthru($image); //dump the contents of the image
}
```



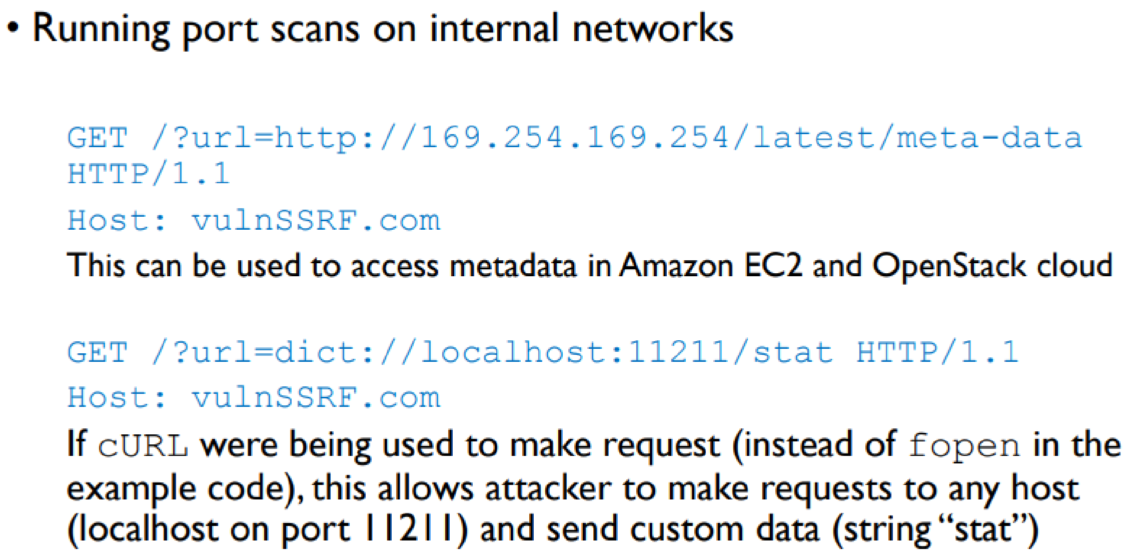
- The attacker has full control of the `url` parameter
- He is able to make arbitrary GET requests to any website and also to resources on the server
  - E.g., by sending these requests to vulnSSRF.com:

```
GET /?url=http://localhost/server-status HTTP/1.1
Host: vulnSSRF.com
```

This returns server status (e.g., services running on the server) information to the attacker

```
GET /?url=file///etc/passwd HTTP/1.1
Host: vulnSSRF.com
```

This allows attacker to access files on the local system



- Running port scans on internal networks

```
GET /?url=http://169.254.169.254/latest/meta-data
HTTP/1.1
Host: vulnSSRF.com
```

This can be used to access metadata in Amazon EC2 and OpenStack cloud

```
GET /?url=dict://localhost:11211/stat HTTP/1.1
Host: vulnSSRF.com
```

If `cURL` were being used to make request (instead of `fopen` in the example code), this allows attacker to make requests to any host (localhost on port 11211) and send custom data (string "stat")

Defense

- Input validation
  - Use a whitelist of DNS name or IP address which your application needs access to
- Response hearing
  - Ensure that expected response is received from the remote server after application sends the request
- Disable unused URL schemas
  - e.g. if only `HTTPS` or `HTTP` schemas are used to make requests, disable other schemas such as `file:///`
- Authentication on internal services
  - Some services do no require authentication by default, enable wherever possible