NANYANG TECHNOLOGICAL UNIVERSITY

Code Injection on the Web (Part I)

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

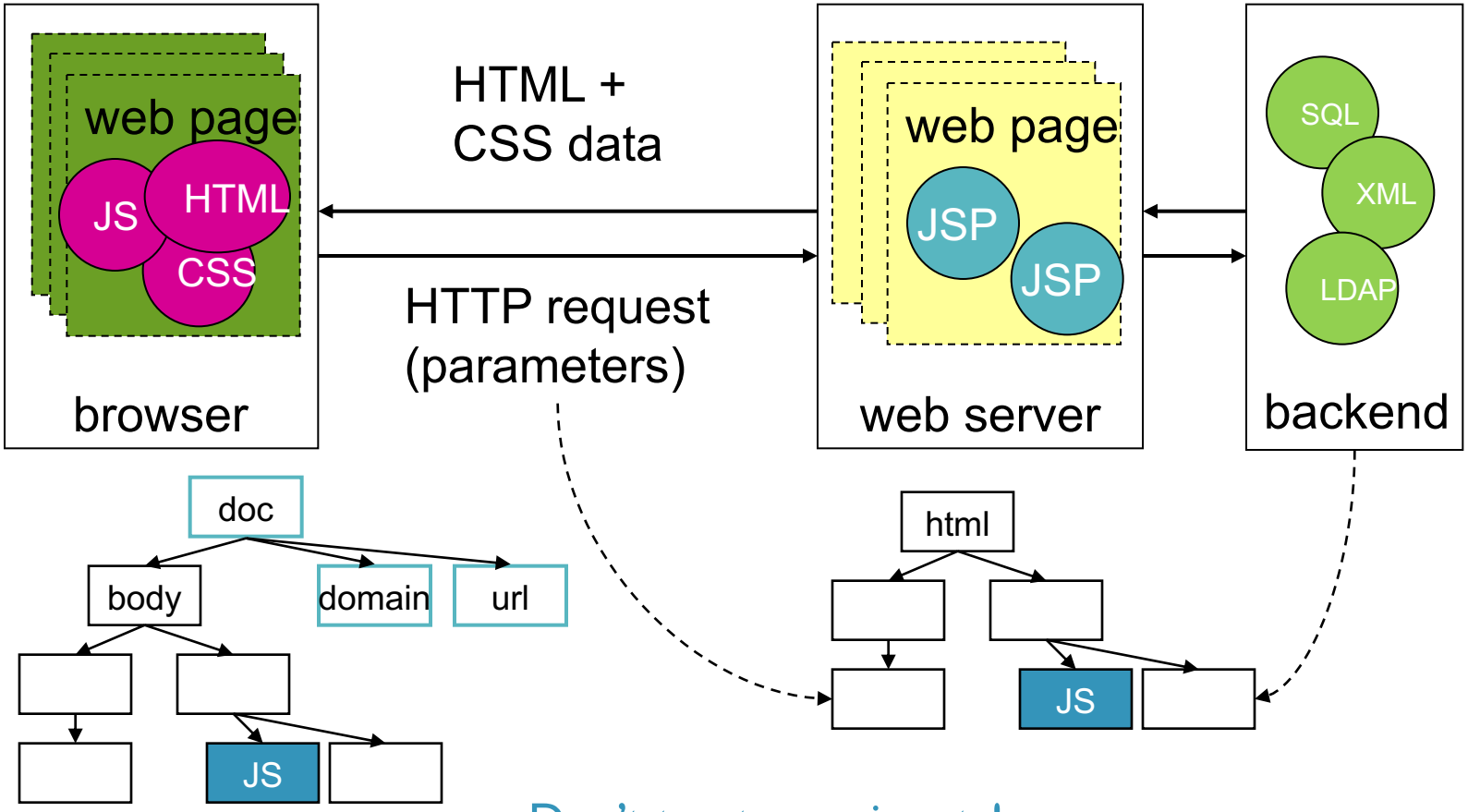
Introduction

- We will now examine software components that dynamically generate interpreted programs from predefined instruction fragments and user input
- Scripting languages support this programming style
  - JavaScript, Perl, JSP, PHP, Python, VBScript, …
- To understand concrete vulnerabilities, attacks, and defences, you must know your scripting language in detail
- Not necessary for understanding the general problem

Dynamic Web Applications

- Web applications are typically constructed using server-side scripting and client-side scripting
- Intentionally malformed inputs may then change the logic of a script
- Code injections are part of OWASP Top Ten, #1
- Don't trust your inputs!

Dynamic Web Applications



Don't trust your inputs!

Java Servlet and JSP

Java Servlet:

```
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // Set response content type
        response.setContentType("text/html");
        // Actual logic goes here
        PrintWriterout=response.getWriter();
        out.println("<html><body>Hello!<br/>");
        out.println("Welcome " + request.getParameter("Name"));
}
```

JSP:

```
<html>
<body>Hello!<br/>
<%
        out.println("Welcome " + request.getParameter("Name"));
%>
</body>
</html>
```

Agenda

- SQL injection
- Writing Secure Code
- Not very well-known but equally serious code injections
  - XPath injection
  - LDAP injection
  - XML injection
- Server-side Request Forgery (SSRF)
- Conclusions

SQL Injection

Attacking user, consumer, corporate data stored in relational databases

SQL Injection

- SQL: Standard language for accessing relational databases
  - Note: Single quote is the string delimiter in SQL
- In a web application, a server-side script (e.g., written in JSP) may construct SQL query as a string from query fragments and user inputs
- String passed to DBMS and executed as SQL query
- User input can now be interpreted as code by DBMS!
- Broken abstraction
  - No clear distinction between data and code

Trusting the Inputs

- How many of you have constructed SQL query like this?

```
1. String pwd = request.getParameter("password");
2. String q = "SELECT * FROM users WHERE Passwd='" +
                                        pwd + "'";

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

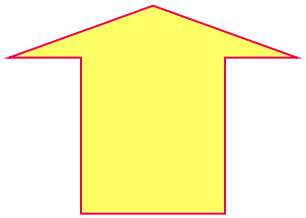
- Which line of code (LOC) makes it vulnerable and should be patched?

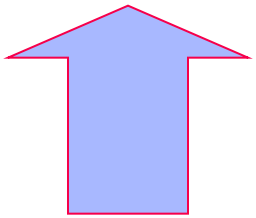
Slide courtesy of Martin Johns

SQL Injection: Programmer's view

Intended by programmer:

```
String pwd = request.getParameter("password");
String q = "SELECT * FROM Users WHERE Passwd = '" + pwd + "'";
```





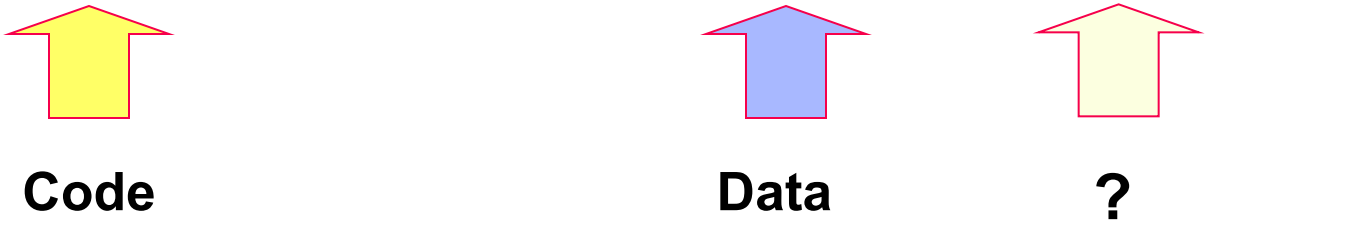
Code **Data**

Slide courtesy of Martin Johns

SQL Injection: DB's view

- Interpreted by the external entity (i.e., the DB):

```
String pwd = request.getParameter("password");
String q = "SELECT * FROM Users WHERE Passwd = '" + pwd + "'";
```



- Determining the semantics of the dynamic input at compile time is in general impossible

Slide courtesy of Martin Johns

SQL Injection: DB's view

- Interpreted by the external entity (i.e., the DB):

```
String pwd = "' OR 1=1";
String q = "SELECT * FROM Users WHERE Passwd = '' OR '1' = '1'";
```

Code **Data** **Code**

- Attack returns entire Users relation because
  - 1 = 1 evaluates to TRUE
  - (Passwd = ' ' OR TRUE) evaluates to TRUE
- Attack pattern for inserting additional SQL command:
  - Enter as password: xyz'; drop table Users;

Poor Defense #1: Simply Getting rid of single quotes

- Attempts to fix one specific problem: malign input terminates string to cut off rest of intended command or paste in new command
- Remove single quotes from inputs
  - But they could be part of legitimate inputs; e.g. name=O'Neill
- Escape with a backslash (or replace single quotes with double quotes)
- Helps in some places:
  - SELECT * FROM client WHERE ID = 'Bob\' or 1=1 -- or salary = 5000'

Example: Escaping with a backslash

```
String user = request.getParameter("user").replace("'", "\'");

String pwd = request.getParameter("password").replace("'", "\'");

String q = "SELECT * FROM users WHERE User='" + user +
                                "' AND Passwd = " + pwd + ";";

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

- Quiz
  - Does this work? → user='Jack'' or 1=1 --'
  - How about this? → pwd=123; shutdown --

Single Quotes are NOT the Only Meta-characters

- Single quote is the string delimiter
- In database, there are other delimiters for other data types
- Integer
  - `SELECT * FROM client WHERE ID=1234 or 1=1 --`
- Note: The comment operator (--) comments out any characters added to the query by the code. Some databases use --, and others use #. Make sure you know the comment operators for the databases you query.

Poor Defense #2: Looking for predefined attack patterns

- The examples use **the classic "1=1" attack**
- Network admins tend to look for that in their intrusion detection systems or firewalls
- We can use something different that circumvent this:

' or 2>1

- There are many tautologies. Just as effective!

Tautology Attack

```
String pwd = request.getParameter("password").
                                        replace("=", "");

String date = request.getParameter("date").
                                        replace("=", "");

String q = "SELECT * FROM users WHERE Passwd = '" + pwd +
                        "' AND Date = '" + date + "'";

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

Quiz: Will this work?

password = rubbish

date = anything' or '1'='1

```
SELECT * FROM users WHERE Passwd = 'rubbish'

                    AND Date = 'anything' or '1''1'
```

Tautology Attack

```
String pwd = request.getParameter("password").
                                        replace("=", "");

String date = request.getParameter("date").
                                        replace("=", "");

String q = "SELECT * FROM users WHERE Passwd = '" + pwd +
                          "' AND Date = '" + date + "'";

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

Quiz: How about this?

```
password = rubbish
date = anything# or 2>1 --
```

```
SELECT * FROM users WHERE Passwd = 'rubbish'
                      AND Date = 'anything# or 2>1 --'
```

Bad Practice: Leaving the Front Door Open

- Sometimes you don't even need a SQL injection vulnerability to compromise the data – When a sysadmin allows attackers to enter the front door by opening the database port such as
  - Microsoft SQL server: TCP/1433
  - Oracle: TCP/1521
  - IBM DB2: TCP/523
  - MySQL: TCP/3306
- Leaving these ports open to the Internet and using a default sysadmin database account password

*You don't need to memorize the port numbers

Writing Secure Code

To prevent code injections

Reference: Michael Howard & David LeBlanc: Writing Secure Code, Microsoft Press, 2nd edition, 2002

Defense #1: Use Prepared Statements

- Instead of building SQL strings dynamically in code, leave the completion of the SQL string to the database
- Tell the database which part of SQL is 'data' not 'code'
- Prepared statements, bound parameters, parameterized commands
    1. Script first compiled with placeholders instead of user input
    2. Replace placeholders by the actual user input when executing the compiled script
- Design principle: change mode of invocation

Example: Use Prepared Statements

- 1) Pre-compile query with placeholders; 2) execute query with actual user input:

```
int uid = Integer.parseInt(request.getParameter("userID"));
String pass = request.getParameter("password");
// '?' serves as placeholders in prepared statement
String query = "SELECT bal FROM account WHERE id = ? AND pwd = ?";
// precompile the query
PreparedStatement pstmt = connection.prepareStatement(query);

// Replace '?'s in the statement with actual input; argument 1 tells
that the first placeholder is to be replaced; setInt tells that the
input must be Integer
pstmt.setInt(1, uid);  //type safety!!!
pstmt.setString(2, pass);

ResultSet results = pstmt.executeQuery();
```

Defense #2: Use Stored Procedures Securely

- Prepared Statements are useful when the database is accessed from an external application (web service); all SQL code stays within the application
- Stored procedures are defined and stored in the database itself and called from the application
- Same effectiveness in preventing SQL injection (they are both secure when implemented securely)
- Auditors should always look for uses of sp_execute, execute or exec within SQL Server stored procedures (or similar procedures) to ensure that user inputs are parameterized!

Poor Defense #2: Stored Procedures with Dynamic Queries

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

- What is the difference with the previous example?
  - ➢ Query is still dynamically built with user input

Poor Defense #2: Stored Procedures with Dynamic Queries ctd.

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

This doesn’t work →

```
xyz’  or  1 = 1  -- ’ (invalid syntax)
```

But this works →

```
xyz’ insert into client values(1005, ‘Mike’) --
```

Bad Stored Procedure

```
CREATE PROCEDURE sp_MySProc @query varchar(128)
AS
exec(@query)
RETURN
```

- Guess what this code does?
  - Not so common but found in a few applications

Example: Using Stored Procedures 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
}
```

Remarks

- Code injection problem can still re-appear even with bound parameters when the query is still built dynamically with user inputs.

- Getting rid of one attack doesn't mean the code is not vulnerable.

Don't trust inputs anywhere!

Remarks

- Other scripting languages differ in the conventions for marking strings, terminating commands, executing external commands, calling prepared statements, …
  - sp_execute, sp_executesql on Microsoft SQL Server
  - **`$mysqli->prepare`** in PHP
- Defences have to be adapted to the scripting language and the database management system.

Defense #3: Whitelist Input Validation

- Bind variables are not useful for some parts of SQL queries such as the names of tables or columns (Why?)
- In such situations, input validation or query redesign is the most appropriate defense.
- For the names of tables or columns, ideally those values come from the code, and not from user inputs.
- But if inputs are used as table names and column names, then the inputs should be mapped to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query.
- Still, this is a symptom of poor design and a full re-write should be considered if time allows.

Example: 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;
```

- Use Input Validation whenever possible, even when using parameterized queries

Validating the Input

Item numbers in the catalogue are integers:

```
<?PHP
    $item = mysql_real_escape_string($_POST['item']);
    $sql = "SELECT * FROM catalogue
            WHERE item =" . $item;
?>
```

```
<?PHP
    $item = $_POST['item'];
    if(is_numeric($item)){
       $sql = "SELECT * FROM catalogue
                       WHERE item =" . $item;
       } else {
       echo "Invalid item numbers!";
       }
   ?>
```

Input Validation using Regular Expressions

- Default-deny regular expression that includes only the type of characters you want

- Only numbers:

```
Pattern.compile("^[0-9]+$").matches(input)
```

- Only letters and numbers:

```
Pattern.compile("^[a-zA-Z0-9]+$").matches(input)
```

- *this slide will not be tested!

Defense #4: Escaping

- Escaping cannot guarantee it will prevent all SQL Injection in all situations
  - e.g. when encoding attacks are possible or when inputs are used in LIKE clauses
- Use it when parameterized commands are not feasible.
- Use it to complement input validation
- Escaping can be database specific in its implementation

Know Where to Patch/Escape

- Quiz

Which line of code (LOC) makes it vulnerable and should be patched?

```
1. String pwd = request.getParameter("password");
2. String q = "SELECT * FROM users WHERE Passwd='" +
                                    pwd + "'";
3. Resultset rs = stmt.executeQuery(q);
```

Know Where to Patch/Escape

- Which line of code (LOC) makes it vulnerable and should be patched?

```
   Codec ORACLE_CODEC = new OracleCodec();
1. String pwd = request.getParameter("password");
2. String q = "SELECT * FROM users WHERE Passwd='" +
       ESAPI.encoder().encodeForSQL(ORACLE_CODEC, pwd) + "'";
3. Resultset rs = stmt.executeQuery(q);
```

- **Quiz: Can we apply the patch at Line 1 or Line 3?**

Escaping Functions

- Java: ESAPI from OWASP (org.owasp.esapi)
  - Available for PHP, Python, Ruby, JavaScript as well
  - E.g. `ESAPI.encodeForSQL(new MySQLCodec(), input)`

- PHP:
  - `addslashes()` adds backslashes (\) to `\,', ", NULL`
  - `mysql_real_escape_string()` escapes certain characters: `\x00, \n, \r, \,', ", \x1a`
- Caution: Always look out for updates
  - `mysql_real_escape_string()` came out PHP 4.3.0, deprecated in PHP 5.5.0
  - New library: `mysqli_real_escape_string()`

Escaping WildCard Characters

- The LIKE keyword in SQL is used for searching a specified pattern
- Two wildcards used in conjunction with the LIKE operator:
  - % :The percent sign represents zero, one, or multiple characters
  - _ :The underscore represents a single character
- These characters must be escaped in LIKE clause criteria (`mysql_real_escape_string()` does not escape this!)
- For example:

```
String input = request.getParameter("input);
input = input.replace("_", "\_");
input = input.replace("%", "\%", $input);
q="SELECT * FROM emp WHERE id LIKE '_" + input + "_'" ;
```

Defense in Depth

- Apply least privilege principle
- Minimize attack surface
- Code review and taint analysis (to be discussed in Week 10)
- Security testing (to be discussed in Week 12)
- Encrypt the stored data – in case of breach
- Do not expose too much information in case of errors or exceptions

Defense In Depth: Least Privilege

- Do not connect database as sysadmin from applications
- Though everything just 'works' when you do it this way, it is very dangerous. Because an attacker may be able to perform any task sysadmin can:
  - Delete any database or table in the system
  - Change any stored procedure, trigger, or rule
  - Delete logs
  - Add new database users to the system
  - etc.
- Most Web-based applications do not need the capabilities of sysadmin to run

Defense In Depth: Least Privilege ctd.

- **Access Control:** determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away.
  - Make sure that accounts that only need read access are only granted read access to the tables they need access to.
  - If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data .
  - Rarely, if ever, grant create or delete access to database accounts.
  - Restrict application accounts to only be able to execute the stored procedures they need.
  - Minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! For example, MySQL runs as system on Windows by default!

Defense In Depth: Be cautious with sending error/exception messages

- Avoid verbose error messages: error messages for invalid inputs can leak information about database, relations, names of columns, etc.
  - Attack can proceed in steps, first find out about the structure of the database, then extract sensitive data
  - Use UNION to get data even out of tables not accessed by the vulnerable script
- Use proper error/exception handling code:

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

https://www.javatpoint.com/example-to-connect-to-the-mysql-database

Quiz: What's wrong with this code?

```
try {

   Class.forName(“com.mysql.jdbc.Driver”);
   Connection con=DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/ntuu","root","root");


   Statement stmt=con.createStatement();
   ResultSet rs=stmt.executeQuery("select * from emp");
                        … … …
} catch (Exception e) {
       System.out.println(e);
}
```

Quiz: What's wrong with this code?

- Catch exceptions appropriately and return controlled error messages (unless you are debugging)

```
try {
   Class.forName("com.mysql.jdbc.Driver");
   Connection con=DriverManager.getConnection(
             "jdbc:mysql://localhost:3306/ntuu","root","root");
} catch (ClassNotFoundException e) {
   System.out.println("Cannot find the DB driver in the classpath!");
}

try {
   Statement stmt=con.createStatement();
   ResultSet rs=stmt.executeQuery("select * from emp");
   … … …
} catch (…) {
        //return controlled error messages
}
```

XPath Injection

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

XML Path Language (XPath)

- XPath can be used to **navigate through elements and attributes** in an XML document.
- Uses "path like" syntax to identify and navigate nodes in an XML document
  - Very much like the path expressions you use with traditional computer file systems:

```
/books/book/title/...
```

- XPath expressions are supported by many languages – JavaScript, Java, PHP, Python, C/C++, etc.

XPath Terminology

- Elements
  - Node, @attribute, '//' anywhere, '.' Current, '..' Parent, '*' wildcard
- Functions
  - name, count, string-length, translate, concat, contains, substring
- Operators
  - + - / *, div, =, !=, <, <=, >, >=, [ ], or, and, mod, | as a union operator

Some XPath Query Examples

- All the titles: /bookstore/book/title
- The title of the first book: /bookstore/book[1]/title
- Titles of the books with price> 35: /bookstore/book[price>35]/title
- Titles of the books in English: /bookstore/book/title[@lang='en']/title
- Title of the book with year=2005 and price=30: /bookstore/book[year=2005 and price=30]/title

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

<bookstore>

<book>
  <title lang="en">Harry Potter</title>
  <price>30</price>
  <year>2005</year>
</book>

<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
  <year>2001</year>
</book>


… … …

</bookstore>
```

XPath Injection

- XML technologies on the rise, increased number of applications using XPath
- Lack of techniques, tools and cheat sheets
- Just like SQLi, the attack can be conducted by placing meta-characters into the input string,
  - which alters the behavior of the original query by modifying the query logic or bypassing authentication.
- A success attack may allow the attacker to query or navigate an entire XML document (no modification)

How Serious?

- Same risks as SQL Injection
- Similar injection techniques
- Much less awareness
- Only a couple of tools for defending
- Plenty of vulnerable applications (untapped?)
- XPath notation/syntax is implementation independent; there are no different dialects like SQL; which means the attack complexity is low.
- Because there is no level access control, it's possible to get the entire document. Some limitations encountered in SQL injection are not there for XPath injection attacks

Example

```
<students>

<student>
<email>wd@svv.lu</email>
<uid>wd003</uid>
<pwd>300wd</pwd>
</student>

<student>
<email>abf@svv.lu</email>
<uid>abf004</uid>
<pwd>400abf</pwd>
</student>
… … …
</students>
```

```
File db = new File("students.xml");
Document doc = DocumentBuilderFactory.newInstance()
             .newDocumentBuilder().parse(db);
XPath xpath = XPathFactory.newInstance().newXPath();

String uid = escape(req.getParameter("uid"));
String pwd = escape(req.getParameter("pwd"));
String query = "//students/student[uid/text()=" +
             uid + " and pwd/text()='" +
             pwd + "']/email";

NodeList nl = (NodeList) xpath.evaluate(query,doc);
```

uid: foo or 1
pwd: nopwd

…[uid/text()=foo or 1 and pwd/text()= 'nopwd']/email

Demo

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

Defenses

- Same as the techniques to avoid SQL injection
- Best is to use a parameterized XPath interface if available!
- For example, in .NET, an XPath expression built with string concatenation:

```
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);
```

Defenses

- Next best option is to validate and escape (OWASP ESAPI Library)
- Need to be aware of emerging technologies and vulnerabilities

- **Existing Work for Penetration Testing**
  - Various presentations and whitepapers about injection techniques
  - XPath-blind-explorer: Windows binary to perform blind injection
  - xcat.py: Blind XPath injection: https://xcat.readthedocs.io

LDAP Injection

Main Source: http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf

Lightweight Directory Access Protocol (LDAP)

- A protocol for **querying** and modifying **directory services** such as corporate email and telephone
- Store information about users, systems, networks, services, applications, etc.
- A common use of LDAP is to provide a central place for authentication (Single Sign-On Solution) and authorization purposes for users.
- Widely used implementations: Microsoft ADAM (active directory application mode) and OpenLDAP

Typical Scenario for an LDAP-based Web Application

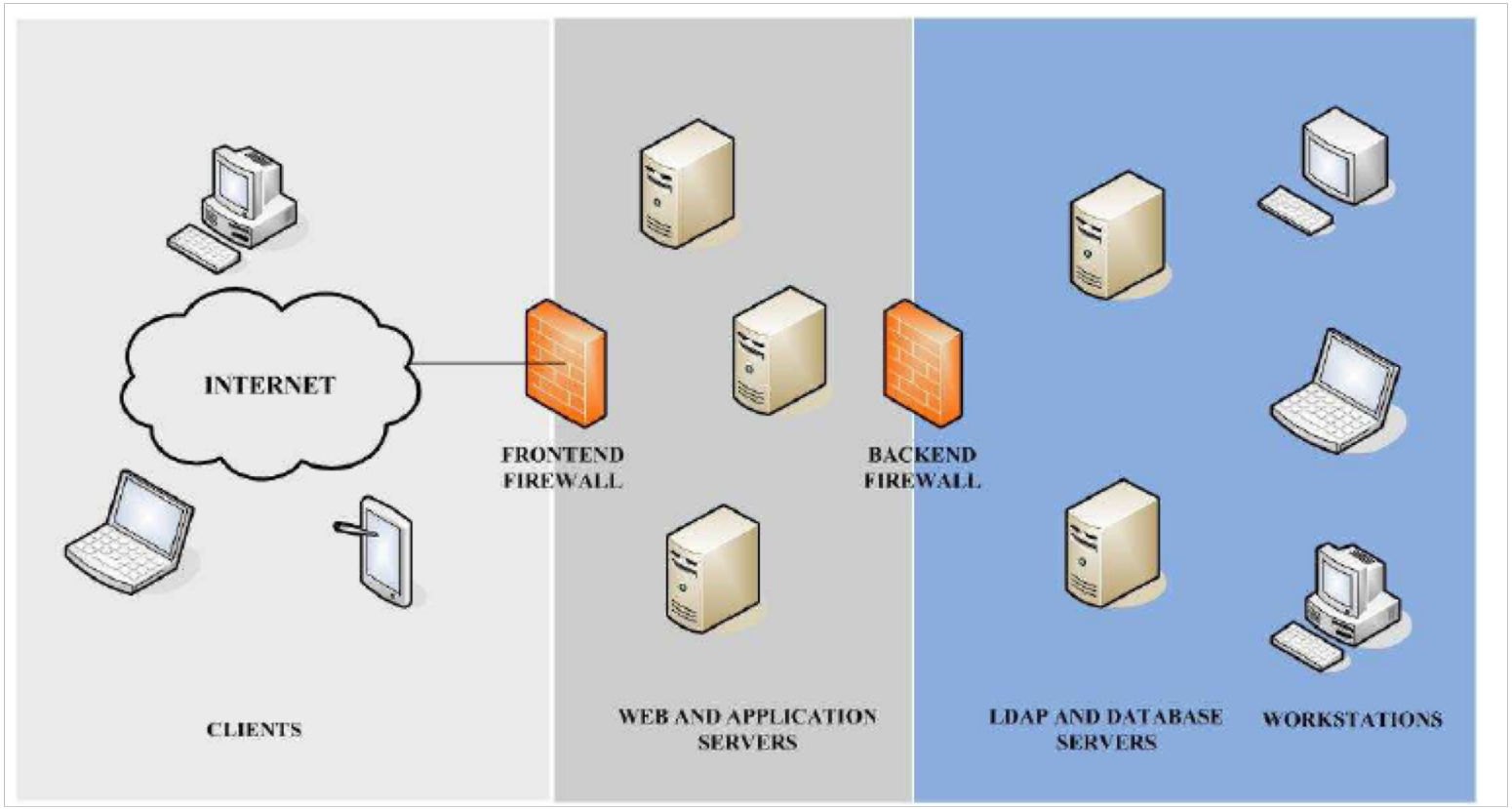
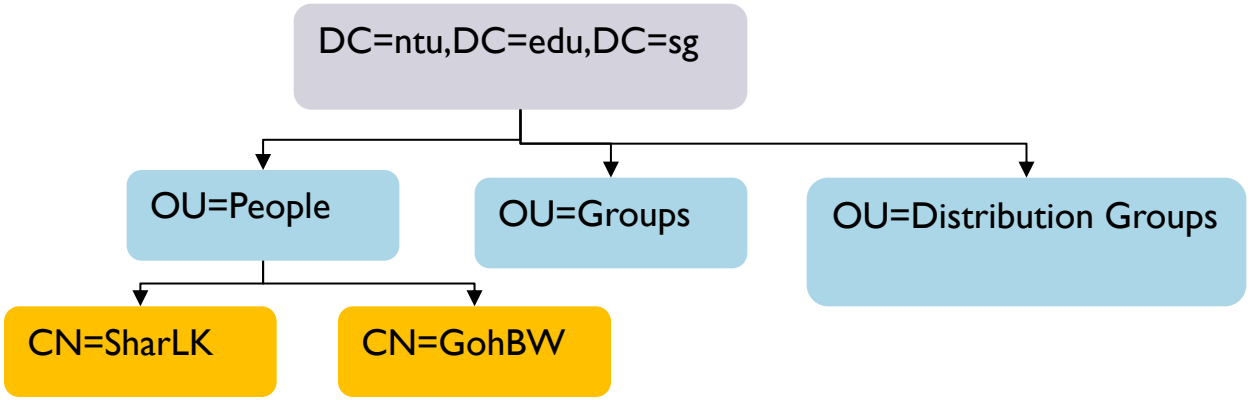


Image courtesy of http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf

A LDAP Tree

- Store and organize information in a tree of directory entries (hierarchical structure) – provides efficient browsing and search capabilities
- Most frequent operation is to search for directory entries using search filters



LDAP Directory Structure

- LDAP uses DNs (distinguished names) as entries in the directory.
- Each DN points to exactly one entry, which can be thought of sort of like a row in a RDBMS.
- Example:

```
dn: cn=SharLK,ou=People,dc=ntu,dc=edu,dc=sg
cn: SharLK
givenName: Lwin Khin
sn: Shar
telephoneNumber: +65 6790 6265
mail: lkshar@ntu.edu.sg
manager: cn=GohBW,ou=People,dc=ntu,dc=edu,dc=sg
```

String X.500 Attribute Type for DN

- CN: commonName
- L: localityName
- ST: stateOrProvinceName
- O: organizationName
- OU: organizationalUnitName
- C: countryName
- STREET: streetAddress
- DC: domainComponent
- UID: userid

You don't need to memorize this!

LDAP Search Filter

- A LDAP search filter is like a SQL SELECT query: Find matches of records stored in a LDAP directory.
- Search operators: logical (&, |, and !) and relational (=,>=,<=, etc.) operators, and special character * that matches one or more characters
- Two 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

LDAP Injection (LDAPi)

- An attack technique that targets programs that build LDAP statements based on user input.
- The attack can be done by manipulating the filters used to search in the directory services,
  - by inserting meta-characters into inputs passed to internal search, add, and modify functions, which alter the logic of the query.
- As a consequence, permissions can be granted for unauthorized queries or for modifying the LDAP trees.
  - Can be critical because the security of many applications rely on single sign-on services based on LDAP directories
- LDAP injection vulnerabilities are common due to:
  - The lack of parameterized LDAP query interfaces
  - The widespread use of LDAP to authenticate users to system

Injection in Search Filters

- (|(attribute=input)(filter2)) or (&(attribute=input)(filter2)):
  - input = "value)(injected_filter)" results in:

    (&(attribute=value)(injected_filter))(filter2))

    - This is syntactically incorrect, but OpenLDAP and some LDAP web clients will ignore the second filter,
    - That is, the attacker doesn't need to pass the second filter
    - Therefore allowing the injection

Injection in Search Filters ctd.

- Some application frameworks will check the filter for syntactical correctness.
- In this case, the injection has to be make sure the (injected) filter is valid:

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

- And the second filter is going to be ignored by the LDAP server

Example: 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

Injecting a wildcard

- Suppose a LDAP search filter like this:
  `(attribute=input)`
- If the input is "*", the filter is:
  `(attribute=*)`
- which matches every value of the attribute

Example: Elevation of Privileges

- Suppose the system allows users to query any document with low security level with this filter:

```
(&(directory=input)(security_level=low))
```

- The attacker could inject an input like:

```
input="documents)(security_level=*))(&(directory=documents"
```

➔

```
(&(directory=documents)(security_level=*))(&(directory=documents)(security_level=low))
```

- In this case, security_level is the injected filter

Defense

- Escape all variables using the right escaping function
  - Escaping process is similar to SQLi but need to consider different set of meta-characters
  - C# .NET: `Encoder.LdapFilterEncode` escapes meta-characters for inputs used in the search filter:

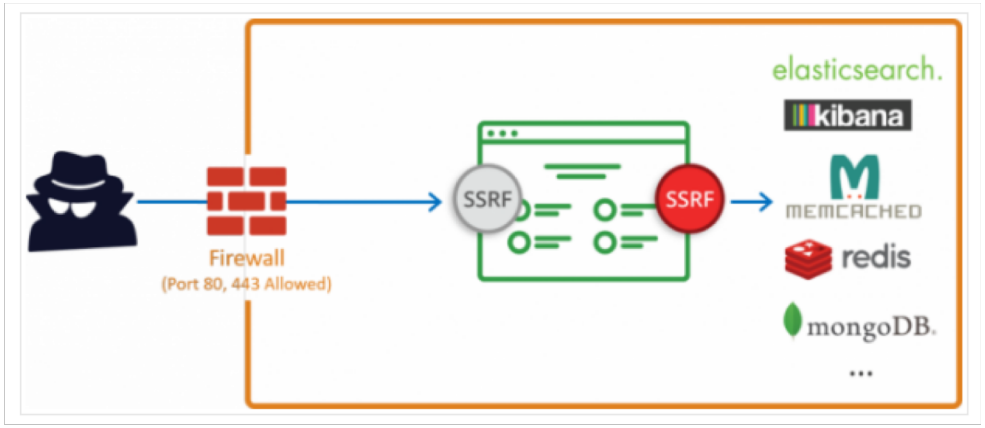
    `& | ! ( ) , \ # + < > ; " = null and leading or trailing spaces`
- Use a framework like LINQtoAD that escapes automatically (https://archive.codeplex.com/?p=linqtoad)
- Defense in depth with usual suspects –
  - Whitelist input validation, least privilege.
  - Do not make LDAP server directly accessible on the Internet (accessible only through applications).

Server Side Request Forgery (SSRF)

https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

Server Side Request Forgery (SSRF)

- SSRF typically occurs when a web application is making a request to internal systems, where an attacker has full or partial control of the request that is being sent.
- A common example is when an attacker can control all or part of the URL to which the web application makes a request to some third-party service.



Why is this bad?

- SSRF can be used for:
  - Scanning other machines within the private network of the vulnerable server that aren't externally accessible.
  - Performing Remote File Inclusion (RFI) attacks.
  - Bypassing firewalls and use the vulnerable server to carry out malicious attacks.
  - Retrieving server files (including /etc/passwd etc).

https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

Example: Code vulnerable to SSRF

```
<?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
}
```

https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

Example: Code vulnerable to SSRF ctd.

- 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

https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

Example: Code vulnerable to SSRF ctd.

- 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 against SSRF

- Input validation: use a whitelist of DNS name or IP address which your application needs access to
- Response handling: ensure that the response received from the remote server (after your application sends the request) is indeed what you are expecting
- Disable unused URL schemas: if your application only makes use of HTTP or HTTPS to make requests, only allow those URL schemas and disable others such as `file:///`, `dict://`, `ftp://` and `gopher://`
- Authentication on internal services: Internally, some services such as MongoDB, Redis do not require authentication by default. Enable authentication wherever possible!

Conclusion

Conclusion

- "A knife sharp enough to cut meet is sharp enough to cut your finger".
- General issue: automatic code generation with inputs from many sources; don't trust your inputs!
- Broken abstraction: separation between data & code.
  - We would need generic rules for analyzing (and modifying) user input for dangerous characters
  - We might trace user input (tainting) on its execution path
- Defences against SQL injection are reasonably mature.
- There are other injections relatively unknown & that still poses challenges.
- Fundamental dilemma: the more dynamic a web page, the more difficult is it to secure the page

DOs and DON'Ts

- Do not trust the user's input!
- Use parameterized queries— not string concatenation— to build queries.
- Be strict about what represents valid input and reject everything else. Use Regular expressions to check valid inputs
- Escape all user inputs
- Do not divulge too much information to the attacker.
- Connect to the database server by using a least-privilege account, not the sysadmin account.

Tutorial 8

Next Week We Discuss…

- **Detecting Injection Vulnerabilities using Taint Analysis**

Look into the following two papers and investigate how a taint analysis-based approach, can detect & localize code injection vulnerabilities:

http://orbilu.uni.lu/bitstream/10993/29045/1/icse2017.pdf

http://people.svv.lu/bianculli/pubs/tsbb-jss2016.pdf